Ph.D. DISSERTATION

Critical-Path-Aware High-Level Synthesis for Fast Timing Closure

빠른 성능조건 만족을 위한 임계경로를 고려하는 상위 수준 합성

BY

Seokhyun Lee

FEBRUARY 2014

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Abstract

Rapid advancement of process technology enables designers to integrate various functions onto a single chip and to realize diverse requirements of customers, but productivity of system designers has improved too slowly to make optimal design in time-to-market. Since designing at higher levels of abstraction reduces the number of design instances to be considered to acquire an optimal design, it improves quality of system as well as reduces design time and cost. High-level synthesis, which maps behavioral description models to register-transfer models, can improve design productivity drastically, and thus, it has been one of the important issues in electronic system level design.

Centralized controllers commonly used in high-level synthesis often require long wires and cause high load capacitance, and that is why critical paths typically occur on paths from controllers to data registers instead of paths from data registers to data registers. However, conventional high-level synthesis has focused on delays within a datapath, making it difficult to solve the timing closure problem during

physical synthesis.

This thesis presents hardware architecture with a distributed controller, which makes the timing closure problem much easier. A novel critical-path-aware high-level synthesis flow is also presented for synthesizing such hardware through datapath partitioning, register binding, and controller optimization. We explore the design space related to the number of partitions, which is an important design parameter for target architecture. According to our experiments, the proposed approach reduces the critical path delay excluding FUs by 29.3% and that including FUs by 10.0%, with 2.2% area overhead on average compared to centralized controller architecture. We also propose two approaches, clock gating and register constrained flow, to alleviate high peak current problem which is caused by the proposed approach. These approaches suppress the peak current overhead to keep it less than 3.6%.

Keywords: High-level synthesis, distributed controller architecture, register binding, controller optimization

Student Number: 2008-30236

Contents

Abstract i

Contents iii

List of Figures vii

List of Tables xi

Chapter 1 Introduction 1

Chapter 2 Background 7

2.1 High-level Synthesis 7

2.2 Subtasks of High-level Synthesis 8

2.2.1 Operation Scheduling and FU Binding 8

2.2.2 Register Binding 10

2.2.3 Controller Synthesis 11

2.2.4 Functional Pipelining Technique for High-level Synthesis 11

2.3 Centralized Controller Architecture 12

2.4 Design Closure Problem in High-level Synthesis 15

2.5 Thesis Contribution 18

Chapter 3 Target Architecture and Overall flow 21

3.1 Target Architecture 21
3.2 Overall flow 23

Chapter 4 Critical-Path-Aware Datapath Partitioning 27

4.1 Introduction 27
4.2 Problem Formulation 30
4.3 Proposed Algorithm 32
4.4 Exploring Design Space for the Number of Partitions 36

Chapter 5 Critical-Path-Aware Register Binding 39

5.1 Introduction 39
5.2 Problem Formulation 40
5.3 Proposed Algorithm 43

Chapter 6 Critical-Path-Aware Controller Optimization 49

6.1 Introduction 49
6.2 Problem Formulation 50
6.3 Proposed Algorithm 55

Chapter 7 Evaluation 63

7.1 Experimental Setup 63
7.2 Design Parameters and Computation Time 66
7.3 Analysis Critical Path Delay on Distributed Controller Architecture 68
7.4 Analysis of Performance and Area 70
7.5 Energy Consumption 78

7.6 Analysis on Register Overhead 80
7.6.1 Clock Gating Approach 81
7.6.2 Register Constrained Approach 84
7.6.3 Combined Approach 86
7.7 Join to Conventional Optimization Techniques on HLS 87
7.8 Comparison with DRFM Binding Approach 87

Chapter 8 Conclusion and Future Work 89

8.1 Summary 89
8.2 Future Work 90

Bibliography 93

Abstract in Korean 103

List of Figures

Figure 2.1 Subtasks of high level synthesis: (a) CDFG representation; (b) scheduled and bound CDFG. 10

Figure 2.2 Conceptual representation of functional pipelining. 12

Figure 2.3 Hardware architecture with a centralized controller. 14

Figure 2.4 Analysis of critical path. 14

Figure 2.5 Design closure problem in HLS. 16

Figure 3.1 Target hardware architecture. 22

Figure 3.2 Overall design flow. 23

Figure 4.1 Architecture graph. 32

Figure 4.2 Updating costs of edges. 34

Figure 4.3 Algorithm structure of datapath partitioning. 35

Figure 4.4 Design space exploration for the number of partitions. 37

Figure 5.1 Motivation of register binding. 42

Figure 5.2 Algorithm structure of register binding. 47

Figure 6.1 Examples of controller optimization. 54

Figure 6.2 Algorithm structure of greedy controller optimization. 57

Figure 6.3 An example with genetic algorithm of controller optimization. 59

Figure 6.4 Algorithm structure of genetic controller optimization. 60

Figure 7.1 Analysis of critical path for distributed controller architecture. 69

Figure 7.2 Comparison results for performance. 71

Figure 7.3 Optimization redundancy of datapath partitioning and controller/MUX optimization. 74

Figure 7.4 Improvement on buffer and register propagation delay. 74

Figure 7.5 Comparison results for area. 75

Figure 7.6 Performance improvement under area constraints. 77

Figure 7.7 Dynamic energy consumption. 79

Figure 7.8 Interconnect length and switching net energy reduction compared to centralized controller architecture. 79

Figure 7.9 Clock network power and peak power consumption. 81

Figure 7.10 Clock gating: (a) peak current overhead from register overhead; (b) peak current reduction using clock gating. 82

Figure 7.11 Reduction of peak power overhead using gated clock. 83

Figure 7.12 Modified flow with register constraint. 84

Figure 7.13 Performance, register area and peak power under data register overhead constraint by 15%. 85

Figure 7.14 Combined approach to reduce peak power overhead. 86

Figure 7.15 Comparison with DRFM. 88

List of Tables

Table 7.1 Resource library (32bit) 64

Table 7.2 Benchmarks details 65

Table 7.3 Design parameters and runtime..... 67

Table 7.4 Information of controller..... 77

Chapter 1

Introduction

Rapid advancement of process technology enables designers to integrate various functions onto a single chip and to realize diverse requirements of customers, but productivity of system designers has improved too slowly to make an optimal design in time-to-market. This problem called as *design productivity gap* [1] makes it important to design in the higher level of abstraction. Since designing at the higher level abstraction reduces the number of design instances to be considered to acquire an optimal design, it improves quality of system as well as reduces design time and cost. Electronic system level (ESL) design to model the entire system with high level languages such as C++ and SystemC [2] has improved design productivity dramatically.

High-level synthesis (HLS), which maps behavioral description models to

register-transfer models, can improve design productivity drastically, and thus, it has been one of the important issues in ESL design. Researches for decades have achieved commercial HLS tools such as CatapultC [3], Cynthesizer [4], and Synphony [5] as well as academic HLS tools. However, the poor quality of synthesis results has been a reason why it has been accepted by the designers for only limited use [6]. It is not unusual to have a large gap between the results of HLS and those of physical synthesis in many aspects, including clock period, area cost, and power dissipation.

The minimum clock period of a netlist can be estimated by the sum of delays of functional units (FUs), multiplexers (MUXs), interconnects, etc. on the critical path. However, the MUX delays and interconnect delays cannot be estimated easily during HLS while FU delay can be obtained before scheduling and binding steps from the library generated by logic synthesis tool. So, it is general practice for the designer to give a design margin in the clock period before the synthesis process, which mostly depends on the designer's intuition. Moreover, as the minimum feature size shrinks, the interconnect delay becomes a more critical issue in modern design[1] since it worsens and makes the design gap between HLS and physical

1 Process variation is also an important issue which can affect the quality of a deep sub-micron design. Although the issue is beyond the scope of our thesis, we expect that our approach can provide better margin to the given delay constraints and help alleviate problems related with process variation.

synthesis unpredictable.

Almost all existing HLS tools generate register-transfer level (RTL) hardware with a centralized controller. Since control signals from the centralized controller typically drive many datapath components through long interconnects, they suffer from longer delay than the signals between datapath components [7]. Hence, the critical path of the synthesized hardware is usually found to be a path from a state register of the controller to a data register. Most conventional HLS tools determine the minimum clock period based on the maximum delay from a data register to a data register rather than from a state register to a data register [26][27][28][29], and it can cause another design gap between HLS and physical synthesis. They consider physical information, but use a centralized controller with an inherent architectural limitation of long wires from the controller to the datapath.

There have been other approaches to HLS, which are based on distributed register architecture [31][32][33][34][35]. The circuits are partitioned into islands such that each island has its own FU(s) and a local register file. Most of the register accesses are to the local register files through short wires, incurring no problems with wire delay. The accesses to register files in other islands are through global wires, which can incur multicycle delay and cause performance degradation. These approaches are effective in reducing the critical path delay in datapath. However, they incur high area cost due to increased number of registers for data copy and

limited register sharing.

In this thesis, we present a novel HLS method using a distributed controller to help speed up timing closure. First, we analyze the impact of a centralized controller on the critical path in a physically synthesized design. Based on the analysis, we propose the use of distributed controller architecture for high-level synthesis. Then, we propose a critical-path-aware datapath partitioning algorithm to reduce the length of interconnects on paths with long delay. It is preferable not to put FUs into different partitions if they are on a potential critical path. A register binding algorithm binds data transfers[2] to registers in order to merge registers on non-critical paths and to split registers on potential critical paths based on partitioning information. Finally, a critical-path-aware controller optimization algorithm distributes the load capacitance driven by registers of the controller such that the critical path delay is reduced.

This thesis is organized as follows. Chapter 2 presents background information of high-level synthesis to understand the proposed method. Chapter 3 proposes a distributed controller architecture and overall design flow for mapping a control data flow graph (CDFG) annotated by HLS to the proposed architecture. Chapter 4,

2 Conventional register binding binds variables—a value is assigned to a variable by a *def* operation and then used by one or more *use* operations—to registers. In the case of multiple *use* operations, it can be beneficial to use multiple registers, one for each data transfer to a *use* operation (or a sub-group of *use* operations). Thus, in this thesis, we use the term register binding to mean binding data transfers to registers.

5, and 6 respectively present three steps of the proposed algorithm: datapath partitioning, register binding, and controller optimization. Chapter 7 shows the experimental results, and Chapter 8 concludes the thesis with comments on future work.

Chapter 2

Background

2.1 High-level Synthesis

HLS (behavioral synthesis or architectural synthesis) implements hardware in RTL from the behavioral model described in high level languages, such as C, C++, and SystemC. Behavioral model, which is input description of HLS, specifies the relation between input and output in algorithm level with variables, operations, and the sequence of operations with control flow. RTL design describes the structure of hardware using FUs, registers, steering logics, and controller. Tasks of HLS implement hardware by assigning operations to FUs and by assigning variables to registers. They also synthesize controller and steering logics to realize sequence of operation with control flow described in behavior model. From many implementation candidates, they try to optimize design under various objectives

and constraints such as area, performance, and power consumption.

2.2 Subtasks of High-level Synthesis

HLS is composed of many subtasks, operation scheduling, FU binding, register binding, and controller synthesis. It can also adopt additional optimization techniques as subtasks to implement an optimal design. Before applying subtasks, it is necessary to transform behavioral model in text to intermediate representation showing data flow and dependency between operations. In this thesis, CDFG, which contains nodes representing operations and edges representing data or control dependencies, is adopted as intermediate representation for HLS as shown in Figure 2.1(a). The first subtask, operation scheduling, determines a control step for each operation to be executed. FU binding selects a FU for each operation among available FUs, and register binding selects a register for a variable or a data transfer to be stored. To control datapath which is generated by previous steps, controller synthesis step synthesizes controller according to scheduling and binding results.

2.2.1 Operation Scheduling and FU Binding

Operation scheduling and FU binding are mapping operations to the temporal domain and to the spatial domain, respectively. Figure 2.1(b) presents scheduling and binding example for given CDFG. Since operations which are scheduled in the

same control step cannot use the same FU and vice versa, scheduling and binding have inter-dependency. Several researches attempt to perform these subtasks simultaneously to find optimal solution [8][9], but those are generally performed independently in many HLS system due to the efficiency of algorithm with time complexity.

Operation scheduling determines when the operation starts. Execution order of operations is determined by the control/data dependency described in CDFG, and operations without dependency can be executed concurrently. Exploiting parallelism of operations can maximize performance but can induce area overhead by concurrently executed FUs. That is, scheduling algorithm explores design space considering trade-off between performance and area. Scheduling problem is known as NP-hard problem [10], but there are many efficient algorithms in terms of the quality of solution and computation time [11][12][13].

FU binding selects FUs to handle operations. As the independent subtask with scheduling, FU binding can be done after or before scheduling. When FU binding is done after scheduling, operations scheduled in the same control step cannot be bound to the same FU. By scheduling and binding, FUs can be shared by multiple operations. It can reduce the number of FUs to execute all the operation in CDFG, but it may cause additional area overhead caused by steering logics such as MUX.

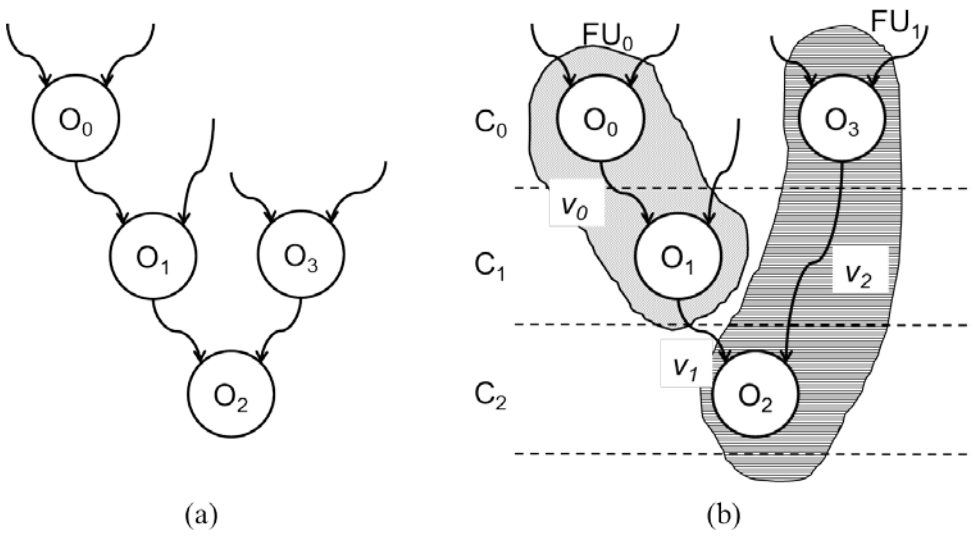


Figure 2.1 Subtasks of high level synthesis: (a) CDFG representation; (b) scheduled and bound CDFG.

2.2.2 Register Binding

After operation scheduling is done, the variable from operation has its own lifetime from the *defined* time to the *used* time. Register binding determines register to store variables during their lifetimes. Since variables of which lifetimes do not overlap each other can share the same register, register binding problem is modeled as graph coloring problem for conflict graph or clique partitioning problem for compatibility graph [11]. In Figure 2.1(b), variable v_0 and v_1 are compatible since lifetimes of them do not overlap, but v_0 and v_2 conflict. Register binding minimizing the number of registers can be easily solved in polynomial time [14][15], but register binding with extra constraints or objectives such as

minimizing MUX or interconnect is known as NP-hard problem [16][17] .

2.2.3 Controller Synthesis

Through subtasks explained previously, HLS implements datapath part of hardware for the given application. Since operation scheduling and FU/register binding make operations and variables to share datapath components, controller is needed in order to forward data to correct FUs or registers. In general, controller is implemented in finite state machine (FSM); control step, external signal or status from datapath, and control signal are represented by state, input, and output of FSM, respectively. Controller synthesis implements controller through general FSM implementation flow, which consists of state minimization, state/output encoding, and logic minimization.

2.2.4 Functional Pipelining Technique for High-level Synthesis

Functional pipelining [11] is an optimization method for generating pipelined circuit to improve the throughput of application. As shown in Figure 2.2, circuit to which functional pipelining is applied starts every initiation interval (II), which is period introducing input data. Functional pipelining may also improve total latency as well as throughput[3]. To apply functional pipelining, scheduling and binding

[3] If functional pipelining is applied, latency of one iteration can increase. However, if it sufficiently iterates many time, total latency can be improved.

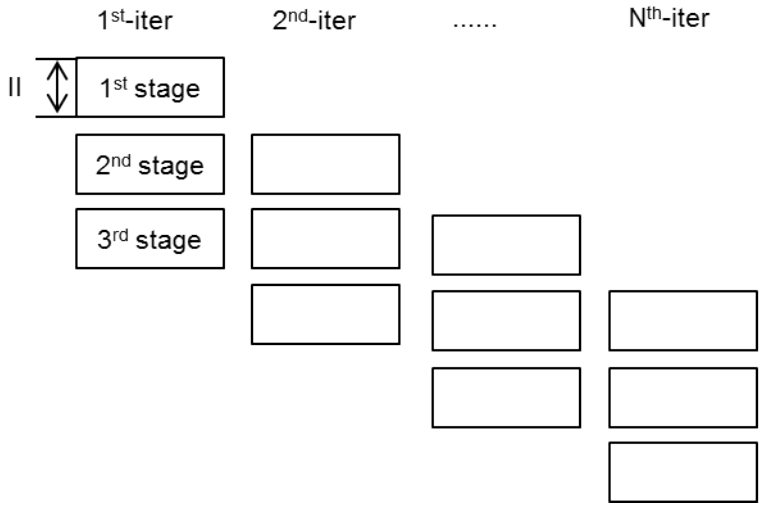


Figure 2.2 Conceptual representation of functional pipelining.

algorithm should be modified since operations in different stages can be executed concurrently. Some extension of heuristic scheduling algorithms [13] can be possible such that they consider parallelism across different stages. Loop pipelining techniques [18][19][20][21] to improve throughput of loop can also be applied to implement functional pipelining in the same manner.

2.3 Centralized Controller Architecture

Figure 2.3 shows the conventional hardware architecture with a centralized controller. It consists of a datapath and a controller. The datapath contains FUs to run arithmetic and logical operations, registers to store data from FUs, and steering logics/interconnects to route data to appropriate modules. The controller gives control signals for correct operation of hardware. It consists of state registers, next

state logics to determine the next state, and output logics to generate control signals. Based on the current state of the controller, control signals select function of FUs to be executed, switch steering logics to route data, and enable registers to store data. The output logic of controller can be implemented in two styles of FSM: non-registered FSM and registered FSM [40]. Registered FSM uses additional registers for controller output signals, while non-registered FSM does not use registers except for state registers.

The critical path lies either on the path from a register in the controller to a data register (p1 in Figure 2.3) or on the path from a data resister to a data register (p2 in Figure 2.3) (note that the delay of controller output logic is removed from p1 if the controller is implemented as a registered FSM. Although conventional HLS tools generally estimate minimum clock period based on the path delays between data registers, the actual critical path usually lies from a register in the controller to a data register. To measure the path delays in centralized controller architecture, we have generated an RTL circuit with twenty multipliers and ten adders from a synthetic example using a conventional HLS flow. Figure 2.4 shows the results of timing analysis of the RTL circuits synthesized with centralized controllers—non-registered and registered—using the TSMC 45 nm technology library. As can be seen from the Figure 2.4, almost all of the top 300 longest paths are from controller to data registers, and among the top 1000 longest paths, only about 7.2% are within

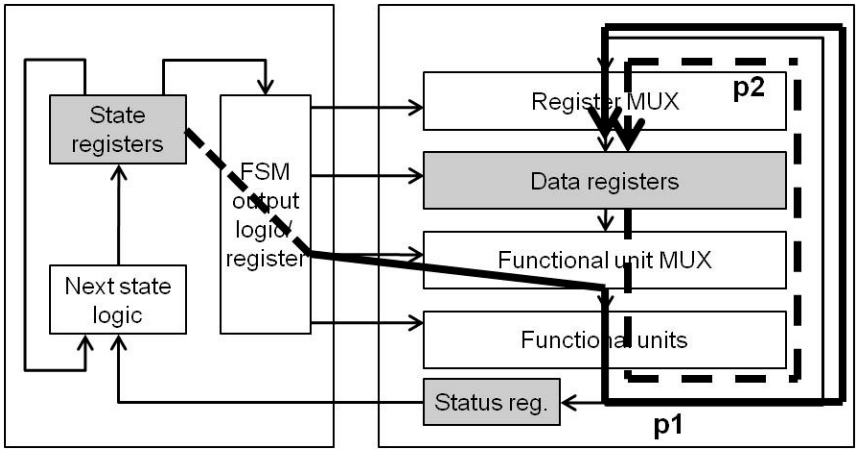


Figure 2.3 Hardware architecture with a centralized controller.

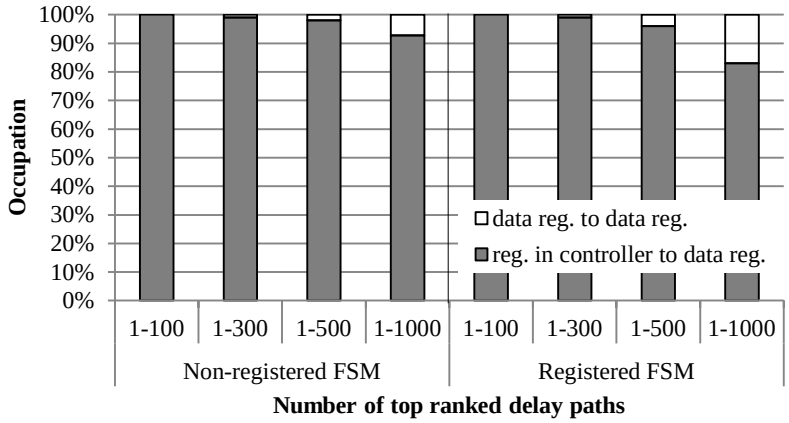


Figure 2.4 Analysis of critical path.

the datapath for the non-registered centralized controller. Even for the registered controller, almost all of the top 300 longest paths are from controller to data registers, and among the top 1000 longest paths, only about 17% are within the datapath.

The centralized controller drives all of the datapath components with control

signals through the output logic of the controller and typically long wires. High fan-out of the controller in this architecture essentially inflicts high load capacitance on the controller, which may cause violations of the rules on maximum transition time and maximum capacitance. To avoid this problem, physical synthesis tools typically apply buffer sizing and/or buffer insertion, which reduces the transition time but adds buffer propagation delay, and also increases the area overhead. This is the main reason why the critical path mostly occurs on a path from a register in the controller rather than from a data register. To overcome this weakness of the centralized controller, a distributed controller is proposed in [7]. It shortens the wires from the controller to the datapath and thus reduces the critical path delay, which is the motivation of our work.

2.4 Design Closure Problem in High-level Synthesis

Conventional hardware design flow using HLS is presented in Figure 2.5. RTL hardware generated by HLS flow from design specification is implemented by logic synthesis and placement/routing tools. It is not easy for designers to know some information before HLS step, such as multiplexer delay and interconnect delay. So, practical HLS flow adopts design margin approach, which gives design margin with predefined value or design margin determined by designer’s intuition considering that uncertain information, when allocating resource used, scheduling, and binding. However, final synthesis results do not meet design constraints with negative slack

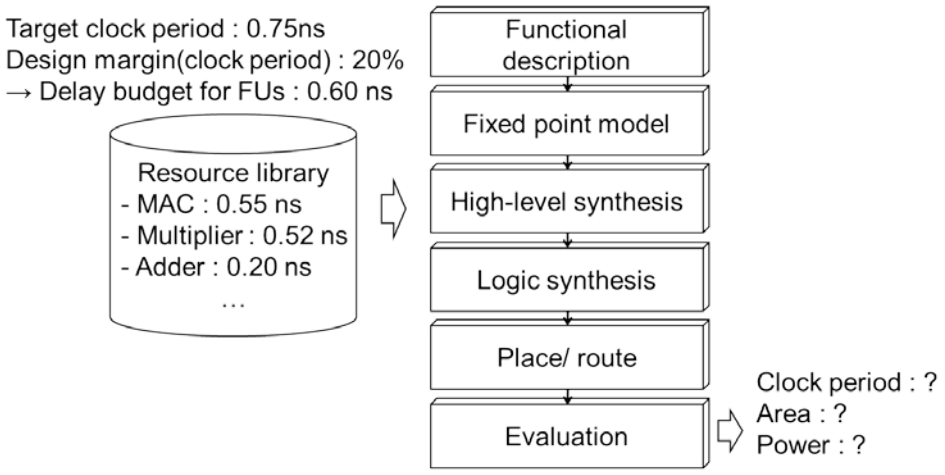


Figure 2.5 Design closure problem in HLS.

although designers have already considered estimation gap at the higher abstraction level. To meet design constraints, when designers can expand design margin, slow FUs such as multiply-and-accumulation (MAC) cannot used for scheduling and binding, and HLS result will be totally different from previous result. So, current result does not also guarantee to meet design constraints, and it causes design closure problem.

Clock period, which is easily violated in hardware design flow, is one of important design constraints to achieve system performance, and its optimization in HLS [22][23][24][25] is important research area to improve quality of results in HLS. To improve system performance, it is important to select clock period to minimize clock slack induced by quantized control step interval with clock period in HLS [22]. [23] proposes operator delay model considering bit-level chaining, and it selects clock period to minimize system latency by reducing clock slack.

Since resource sharing and allocation can affect clock period, delay estimation and clock minimization approach is proposed [24]. Since interconnect delay is an important component determining clock period in current deep sub-micron era, method to consider interconnect delay during optimal clock selection is proposed [25]. Although these approaches improve clock period by considering datapath delay, they overlook the fact that critical path delay of conventional RTL hardware mostly occurs on the path from controller to datapath.

One of important solution to alleviate design closure problem is to reduce the gap between HLS and lower level synthesis (logic synthesis and placement/routing). Considering low level information during HLS can help reduce the gap between HLS and lower level synthesis. However, it may be very time-consuming especially when it considers all the information which is not useful for achieving significant improvement or which is too inaccurate to be estimated at the higher abstraction level. Some techniques have been proposed to consider physical information for high-level synthesis [26][27][28][29]. To use physical information, a simple physical synthesis is applied to RTL generated by HLS. For example, the approach in [26] annotates post-layout delays on the CDFG and then reschedules operations and re-synthesizes the controller for continuous time domain without changing resource binding. The iterative approach in [27] estimates interconnect delays between datapath components through incremental floorplanning and then modifies

the HLS results incrementally. The approach in [28] uses stochastic wire length model to estimate the critical path delay of a datapath and regenerate the datapath iteratively. The approach in [29] takes the global wire reduction technique using idle FUs. It also uses physical information obtained by early placement for initial scheduling and binding. Considering that the critical path of a conventional RTL design often occurs on the path from a state register of the controller to a data register, the aforementioned approaches have limitations since they estimate the path delays only between datapath components and modify scheduling and binding only for the datapath based on that physical information. Since they also consider too much low-level information, they are often time-consuming.

2.5 Thesis Contribution

Main contribution of this thesis is the first to consider the following design aspects to reduce critical path delay:

— Our approach considers all possible critical paths including ones between controllers and data registers as well as ones between data registers. For the reduction of critical path delay, it integrates datapath partitioning, register binding, and controller/MUX encoding based on physical information.

— The datapath partitioning algorithm distributes high capacitance loaded on the centralized controller to distributed local controllers and reduces interconnect delay from controller to datapath.

— The register binding algorithm considers reducing critical path delay whereas conventional register binding algorithms for distributed architecture consider just reducing number of registers [32][34].

— The controller optimization algorithm properly distributes load capacitance over the control signals from controllers to datapaths. This algorithm allows high load capacitance on non-critical paths but reduces it on critical paths (keeping the aggregate load capacitance driven by the controllers unchanged).

— The proposed design flow can be coupled with conventional HLS flows utilizing architectural optimization such as functional pipelining. The architecture that we are targeting is just a small extension of the architecture assumed by the conventional HLS flow, and there is no restriction on applying optimization techniques used in the conventional HLS flow (note that the DRFM approach cannot be integrated easily into conventional HLS flows due to the completely different architectural assumptions). This approach does not restrict the design space for scheduling and binding.

Chapter 3

Target Architecture and Overall flow

3.1 Target Architecture

Our target architecture is obtained by partitioning the conventional centralized controller architecture as shown in Figure 3.1. FUs (multipliers, adders, load/store units for memory operation, etc.), registers, and a controller in the same partition are connected with relatively short wires. Each FU can access registers in other partitions as well as those in the same partition, but interconnects to the registers in other partitions may be longer and thus have longer delay. In this architecture, registers store data from FUs in the same partition, and transfer the stored data to FUs that require them. Additional registers are not added between inter-partition interconnects to preserve architectural consistency with centralized controller architecture; this architecture can utilize the architectural optimization results from

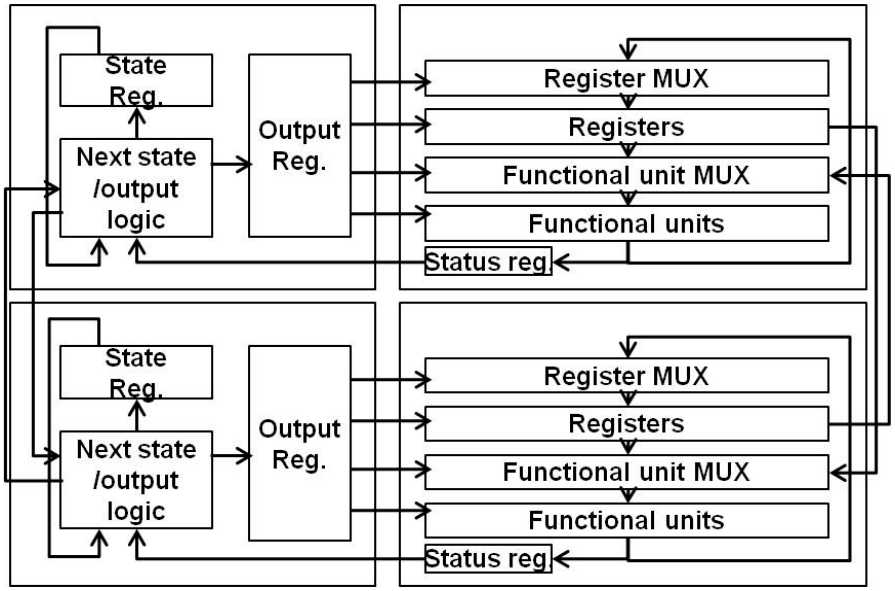


Figure 3.1 Target hardware architecture.

the conventional HLS flow, including functional pipelining, scheduling, and binding. Each partition has its own controller, which drives datapath components in the same partition. Each controller separates an output register from the state register for better output encoding and lower capacitive loading. Also, to reduce the performance gap appearing after physical synthesis, it is necessary to reduce the delay elements such as inter-partition interconnects, MUX, and high load capacitance that are on the combinatorial path containing *critical FUs*[4] such as

[4] We consider FUs with delay larger than 70% of the longest FU delay as *critical FU*s. This threshold is determined based on our assumption that delays of inter-partition interconnects do not exceed 30% of the longest FU delay (typical timing margin in HLS is 20~30%). We also assume that delay variation does not seriously affect the quality of the results due to this delay margin, though it may depend on the quality of the process. Actually, all HLS tools suffer from the same problem and we believe that our approach will effectively alleviate the problem by reducing the potential critical path delays.

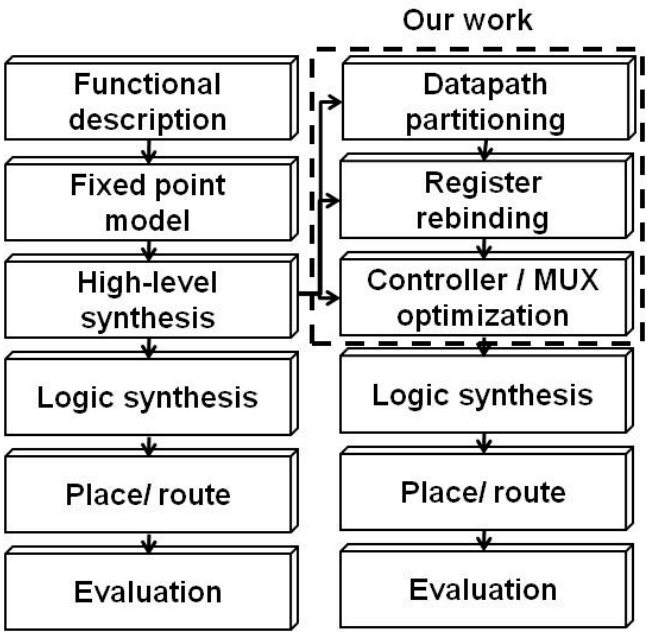


Figure 3.2 Overall design flow.

multipliers.

3.2 Overall flow

Figure 3.2 shows an overview of the proposed design flow. The left side of this figure depicts a conventional design flow including HLS. The designer first modifies the input behavioral/functional description to a fixed point model if it is necessary. Then the conventional HLS flow generates an RTL design from the results of scheduling and binding, and logic synthesis followed by placement and routing generates a layout for evaluation. Our work starts from an intermediate

representation generated by the conventional HLS flow. It is actually a CDFG[5] annotated with the scheduling and binding results. First, we construct a graph representing the RTL structure obtained from the scheduling and binding information without register binding yet; each data transfer uses a dedicated register in this step.

During the datapath partitioning step, datapath components are clustered according to their connections and the controller is replicated such that each cluster gets its own controller. Replicated controllers are synchronized by a single clock without any global controller. They implement FSMs having the same state transitions for the same present state and the same input control signals. However, each controller generates its own output signals to control datapath components in its own partition. Then we perform register binding and controller/MUX optimization to reduce critical path delays by utilizing slacks in non-critical path delays and redistributing capacitive loading to output registers of the controllers. The register binding algorithm allocates data transfers to registers such that delays of MUXs used for sharing registers on the critical paths are reduced. Controller/MUX optimization allocates more load capacitance to the controller

[5] Datapath partitioning and controller/MUX optimization flow can be used regardless of existence of control dependency in the given application since they use an RTL structure generated by the scheduling and binding in the conventional HLS flow. Since the proposed register binding algorithm is devised for general compatibility graphs, it can also be applied to the CDFG.

output registers that are not on the critical paths. We finally generate a partitioned RTL and pipeline it to the next synthesis process.

As we go through the three new design steps (from datapath partitioning down to controller/MUX optimization), more detailed information on physical parameters is used to further optimize the design. In the partitioning step, for example, we consider logic delays of FUs and the relative distance between FUs. In the register binding step, we consider MUX delay and partitioning results. In the controller/MUX optimization step, we consider the additional delay of control signals due to the loading by control inputs of datapath components.

The three steps have forward dependency but do not have backward dependency. For example, datapath partitioning may affect register binding but the other way is not true since the parameters (e.g., FU delays) used for datapath partitioning are not changed by register binding.

Chapter 4

Critical-Path-Aware Datapath Partitioning

4.1 Introduction

As explained in Chapter 2, the conventional centralized controller architecture suffers from high capacitive load to controller and long interconnect delay from controller to datapath. Distributed architecture is a beneficial approach to reducing overall wire length and to improve system performance. Distributed logic-memory architecture in [30] reduces memory access conflicts by partitioning memory and datapath, but it does neither consider wire length nor try to optimize clock period of the system. The approach in [31] combines HLS with placement for distributed register architecture to minimize system latency. It can optimize datapath delay systematically since it isolates communication delay from computation delay.

However, it has limitations in that it does not consider the actual critical path from the controller to data registers. Architectural restrictions such as that accessing registers of different FUs may take several cycles and registers can be shared only by outputs from the same FU can degrade performance and area. Another popular distributed architecture used in HLS is distributed register file microarchitecture (DRFM) [32][33][34][35]. [32] presents a resource binding and interconnect optimization method for DRFM, targeting FPGAs. It shows that the DRFM approach can reduce the clock period and MUX area compared to the conventional architecture. However, it uses register files which limit the number of read/write ports to reduce area and delay overhead. The limitation restricts exploiting parallelism such as functional pipelining. Inflexibility caused by using register files also restricts adding registers to reduce critical path delay. The study focuses only on optimizing the MUX delay in front of data registers and the number of inter-island interconnects that are on paths from data registers to data registers, overlooking the delay from the controller to datapath. The approach in [33] uses a controller distribution technique. However, it is not for optimizing the path delay from state registers to data registers but for reducing controller cost by partial duplication of states. Since DRFM also has the architectural restrictions that FUs can access registers in other islands in multiple cycles and registers can be shared only within an island, it has overhead in performance and area. Although the

generalized DRFM (GDR) proposed in [34] can relax those restrictions, it still has restrictions in scheduling and binding due to the aforementioned architectural restrictions of DRFM. Furthermore, it does not attempt to optimize the local controllers; it leaves the entire controller synthesis task to conventional logic and physical synthesis tools. Another recent distributed register architecture called HDR (huddle-based distributed-register architecture) [35] is divided into non-uniform islands, called huddles. To improve energy efficiency, it assigns a high supply voltage to critical huddles and a low supply voltage to non-critical huddles. It focuses on energy efficiency while our architecture and algorithm focus on improving critical path delay while using a single supply voltage.

In this chapter, we propose datapath partitioning algorithm for proposed distributed controller architecture to distribute capacitive load to controller and reduce interconnect length from controller to datapath. Since interconnect delay across the different partitions may be long, it is necessary for components on the critical path not to be connected with that interconnect. Although partitioning datapath into as many as possible is useful to reduce capacitive load and interconnect, it can cause controller overhead and register overhead which affects clock tree synthesis of lower level synthesis. It can also make following optimization step (register binding and controller/MUX optimization) to lose global information, and the efficiency of those optimizations will be degraded. Proposed

datapath algorithm considers those design aspects, partitioning policy not to connect components on the critical path with inter-partition interconnect and exploring the number of partitions to maximize clock period improvement when proposed algorithms are used.

4.2 Problem Formulation

Datapath partitioning makes datapath components to cluster around distributed local controllers. It shortens interconnect between controllers and datapath components and reduces load capacitance to controllers. To make the partitioning algorithm effective, we need to identify beneficial components to be clustered together.

It is clear that interconnects between different partitions may cause relatively long delay. However, such inter-partition interconnects can avoid being included in the critical path if they are used to connect only FUs with relatively short logic delay. So, we propose a critical-path-aware datapath partitioning algorithm, which performs partitioning such that interconnects that are likely to be in the critical path are not cut by the partitioning. To apply the algorithm, we construct an architecture graph $G_A(V_F, E_C)$ from the FU binding information (initially, registers are not shared), where V_F is a set of vertices, each of which represents an FU and its output register(s), and E_C is a set of directed hyper edges, each of which represents a connection of two or more vertices. An edge connecting more than two vertices

implies that outputs from two or more predecessor vertices are multiplexed to provide data to the successor vertex, and in that case, the edge representation also includes a MUX. In the case where a MUX provides data to two or more successor vertices, the edge is replicated according to the number of successors. Figure 4.1 shows an example of architecture graph. Each edge has its own weight (w) representing the penalty for being cut by partitioning. It is calculated based on the criticality of FU delay and the number of cuts which will be explained in Section 4.3. Then the datapath partitioning problem can be formulated as follows.

Problem 1: Given an architecture graph $G_A(V_F, E_C)$, edge cost function $w: E_C \rightarrow Z^+$ and an integer k, divide the graph into k partitions such that the total cost of edges cut by the partitioning is minimized.

As explained before, long interconnects do not matter if they are used to connect FUs with short logic delay. That is why we include the logic delay of an FU in the cost of an edge. However, we may not be able to avoid cutting some interconnects with high cost. In that case, we can try to place FUs connected by such an interconnect close to each other even if they are in different partitions. This will be possible only when the number of such inter-partition interconnects is small. That is why we set up the objective as minimizing the total cost of inter-partition interconnects.

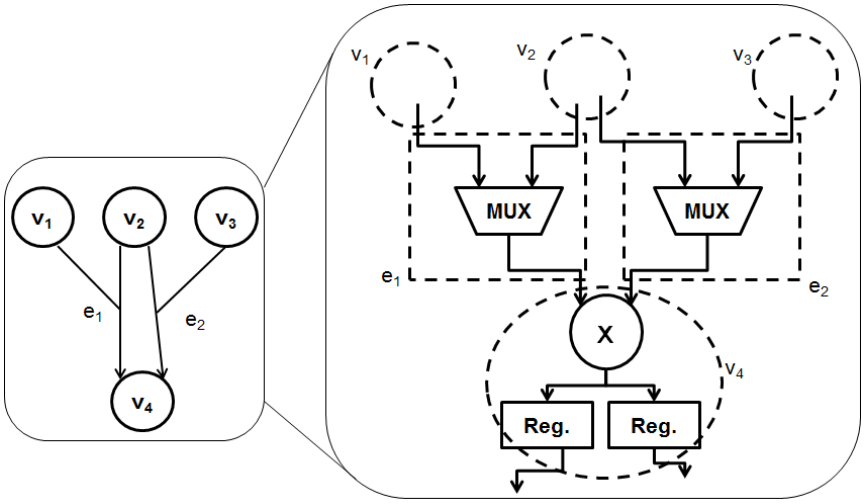


Figure 4.1 Architecture graph.

Minimum cut into bounded sets of a graph, which partitions graph into two sets of vertices such that the sum of weights of edges cut is no more than positive integer K, is known to be NP-complete [10]. The datapath partitioning problem, which minimizes the sum of weights of edges cut, is NP-hard since it is at least as hard as minimum cut into bounded sets problem.

4.3 Proposed Algorithm

To solve Problem 1, we adopt the two-way Fiduccia-Mattheyses (FM) partitioning algorithm [36] and the terminal propagation method [37]. Then k-way partitioning is performed by applying the FM partitioning algorithm iteratively. The number of partitions (k) is determined in such a way that each partition is properly sized (this is based on an empirical observation; refer to Section 4.4 for the details). The

outline of the algorithm is shown in Figure 4.3. It has (*k*-1) iterations of the main loop body for k-way partitioning as shown in line 3. In each iteration, it updates the cost of each edge based on the number of cuts (the more an edge is cut by the iterative partitioning, the longer the corresponding interconnect tends to be), selects a partition to be divided further, and then performs partitioning of the selected partition.

Figure 4.2 shows two cases of updating the cost of edges. In the case of Figure 4.2(a), the edges connect FUs within the same partition. Edge 1 connects vertex A (some FU) to a multiplier, which is a critical FU. So, the edge gets higher cost for partitioning than edge 2. In the case of Figure 4.2(b), the circuit has already been partitioned to some extent, where edges (hyper edges) 3 and 4 are connecting the FUs (C, D, E, and F) in partition P 1 and the FU (B) in partition P2. Assume that partition P1 is to be further divided into smaller partitions in the current iteration. Since edge 3 is connecting FUs in different partitions, we expect that edge 3 will be implemented by a longer interconnect than edge 4. Thus, we assign higher cost to edge 3 so that the edge is less likely to be cut again during the partitioning of P1. This is done in order not to further increase the length of an already long interconnect.

To reflect the concept of cost due to edges cut by partitioning, each edge *e* is assigned with weight *w* given by

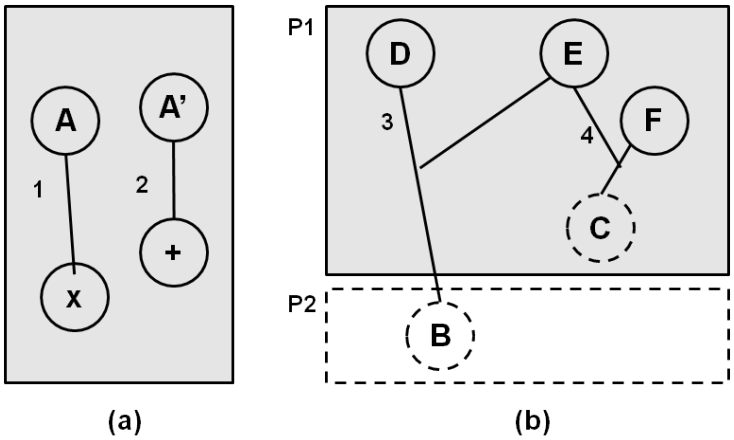


Figure 4.2 Updating costs of edges.

$$w(e)=\alpha*(\#critical\ FUs)+\beta*(\#cuts) \quad (1)$$

Thus the weight of an edge is proportional to the number of *critical FU*s connected by the edge and the number of cuts made on the edge (the number of cuts is assumed to be the same as the number of partitions that the corresponding interconnect should span). The edge weight is calculated by procedures InitialEdgeCost and UpdateEdgeCost in Figure 4.3. Line 5 of InitialEdgeCost calculates the left-side of the addition in (1), and line 3 of UpdateEdgeCost calculates the right-side.

We set α=2 and β=1, which are determined empirically for a rough estimation of the relative delay since it is good enough at this step. More refined delay estimation will be used in the following steps.

```
DatapathPartition(G_A(V_F, E_C), w, k)
1   F_p(1)←V_F, F_p(k) is k^th partition
2   InitialEdgeCost(V_F, E_C, w)
3   for i ← 2 to k {
4       UpdateEdgeCost(V_F, E_C, w)
5       sel = SelectParition(i, E_C, w, {F_p(j)| j = 1...i-1})
6       FMPartition(F_p(sel), E_C, w, F_p(i))
7   }
InitialEdgeCost(V_F, E_C, w)
1   for c in E_C {
2       w(c)←0
3       for f in F_c, F_c is the set of FUs connected to c∈E_C {
4           if f is critical module
5               w(c)←w(c)+α
6       }
7   }
UpdateEdgeCost(V_F, E_C, w)
1   for c in E_C {
2       if c is cut during the previous partitioning
3           w(c) ←w(c)+β
4   }
SelectPartition(i, E_C, w, {F_p(j)| j = 1…i-1})
1   maxsize←0
2   for j←1 to i-1 {
3       if (|F_p(j)| ≥ 2)  ∧  (maxsize < size of F_p(j)) {
4           maxsize←F_p(j)
5           maxpartition←j
6       }
7   }
8   return maxpartition
```

Figure 4.3 Algorithm structure of datapath partitioning.

To further divide the design into more partitions, procedure SelectPartition in Figure 4.3 selects a partition (having two or more FUs) that has the largest area. This is to obtain a well-balanced partitioning result.

The FM partitioning algorithm is run (k-1) times as shown in Figure 4.3. Since each partition contains at least one FU, the number of partitions k cannot exceed the number of FUs, and thus, k=O($|V_F|$). The complexity of the FM algorithm is O($|P|$), where $|P|$ is the total number of pins [36]. UpdateEdgeCost checks to see if each edge is cut during the previous partitioning and thus has a complexity of (O($|E_C|$). SelectPartition takes the summation of edge costs for each partition and selects the one with minimum cost, which takes O($|P|$) time. Thus, the complexity of the entire algorithm is O($|V_F||P|$).

4.4 Exploring Design Space for the Number of Partitions

The main purpose of partitioning is to reduce the criticality of global interconnects and to distribute capacitance loaded on the controller. If the area of a partition is large, we may not achieve sufficient reduction of interconnect delays and load capacitance. On the other hand, if the area of a partition becomes too small, the optimization of register binding and the controller is limited since global information for optimization is lost. Area overhead also increases since sharing data registers and controller output registers is restricted within the boundary of a partition.

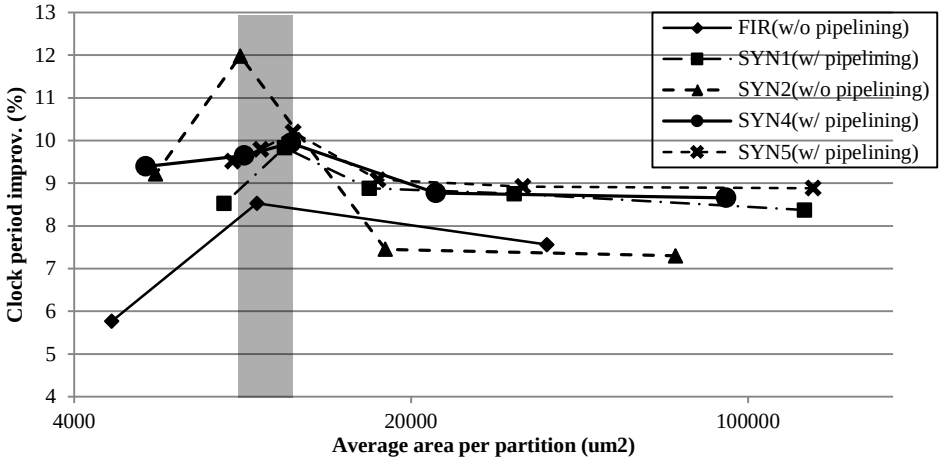


Figure 4.4 Design space exploration for the number of partitions.

Figure 4.4 shows critical path delay improvement for different values of area-per-partition (total area divided by parameter k). In case of SYN2, for example, when k is set to 1, 4, 8, or 12 (corresponding area values in x axis are 70793, 17698, 8849, and 5899), critical path delay improvements in y axis are 7.3, 7.5, 12.0, and 9.2 percent, respectively. From those empirical results, we assume that critical path delay improvement is maximized when we set k to a value in that area bucket of $0.8\sim1.2\times10^4$ um^2. If we increase the area beyond this range, the delay due to intra-partition interconnect is no longer ignorable according to the parameters of metal layer from the TSMC library[6]. So, we have determined the number of partitions

[6] For example, considering that the effective resistance of a 2:1 MUX cell is 2,800Ω, the increase of delay due to load capacitance is estimated by 2,800Ω×(load cap.)pF [47][50]. Since the interconnect load is about 0.007pF/100um according to the library, the additional delay due to the 100um interconnect will be about 19.6ps, which is comparable with the delay of the MUX cell (20ps as shown in Table 7.1).

such that the average area of a partition is in this range.

Chapter 5

Critical-Path-Aware Register Binding

5.1 Introduction

Register binding is traditional subtask of HLS. Initially, each variable can use its own register, but register sharing is necessary because of register overhead. Since registers shared by several variables may inflict input MUXs, register sharing reduces register area at the cost of clock period. However, if operations which produce variables sharing the same register use the same FU, register sharing reduces area without the cost of MUX. Conventional register binding explores those design aspects and tries to reduce the number of MUXs[7] or the area of MUXs.

[16] proposes register binding algorithm to minimize the number of MUXs. It

[7] The number of MUX is generally acquired by modeling MUXs to the trees of 2:1 MUXs.

also considers port assignment of FUs which also affects the number of MUXs. Since FU binding result affects the quality of register binding result (note that the number of MUXs does not increase if operations which produce variables sharing the same register use the same FU.), simultaneous FU and register binding [17] reduces the area of MUXs. However, since MUXs on the non-critical path do not increase clock period, minimizing the number or the area of MUXs is insufficient to optimize clock period.

In this chapter, we propose a heuristic register binding algorithm to optimize clock period. Motivated by the fact that MUXs on the non-critical path do not increase clock period, it tries to share data transfers on the non-critical path as much as possible. Data transfers who pass through the critical path do not share the same register or share register only when sharing does not inflict MUX.

5.2 Problem Formulation

Based on the result of the datapath partitioning algorithm, we bind registers used for data transfer. Figure 5.1 shows a motivational example of the register binding. In the initial binding, each data transfer is assigned with its own register as shown in Figure 5.1(a). This binding can provide the minimum achievable delay since there is no MUX used for sharing a register, but it is area-inefficient. Figure 5.1(b) shows a typical register binding (only one shared register is used) obtained when the data transfers, v_{AC}, v_{BD}, and v_{BE}, are compatible (i.e., there is no overlap of live

ranges of the corresponding data transfers, or the three data transfers are for a single variable). However, such sharing may require MUXs and/or long interconnects, and it may worsen the critical path delay of the circuit. One of register binding policies to avoid this is to split a register that sends data to multiple partitions such that each register drives only FUs in one partition [38]. In our example, the policy generates the circuit shown in Figure 5.1(c). It can reduce the interconnect length associated with v_{BD}. However, this policy cannot sufficiently explore the design space associated with register binding. For example,

1) If the path to FU D is not a critical path, Reg1 added in Figure 5.1(c) can be an unnecessary overhead.
2) If the path to FU C is on a critical path, register binding shown in Figure 5.1(d) can reduce the critical path delay more effectively by removing the MUX in front of Reg0.

Thus, to explore the design space, we devise a register binding algorithm to use shared registers on the non-critical paths and dedicated registers on critical paths. The register binding problem can be formulated as follows.

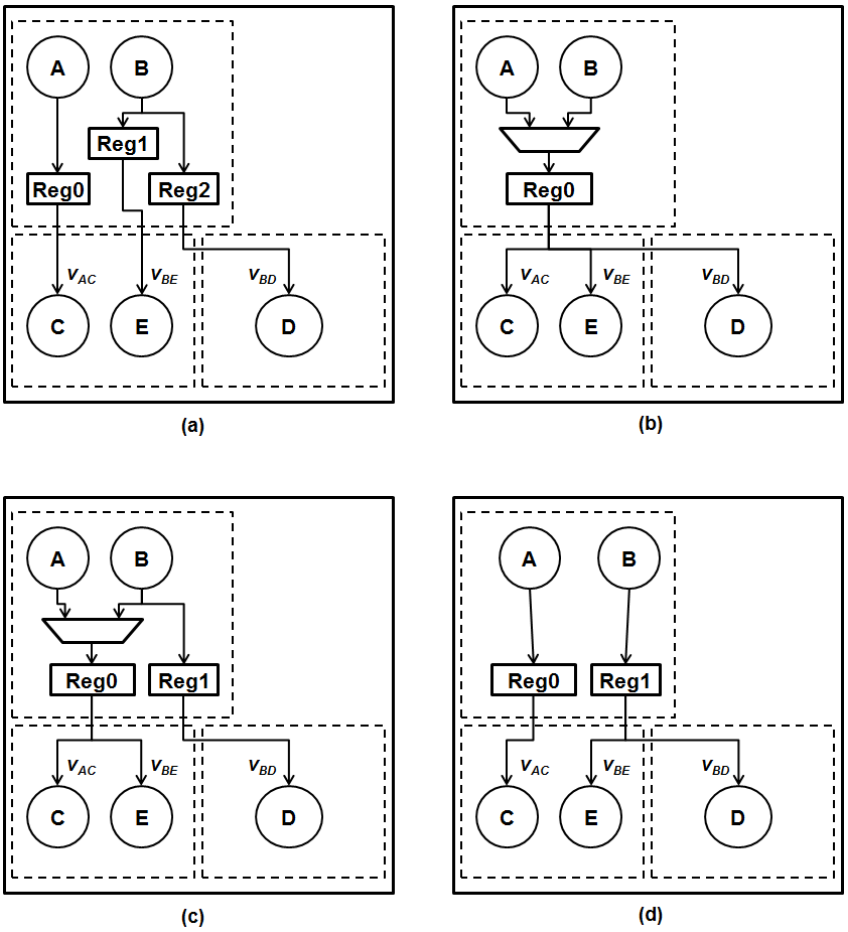


Figure 5.1 Motivation of register binding.

Problem 2: Given a control data flow graph $G_{CDF}(V_O, E_O)$, an FU binding $\pi: V_O \rightarrow F$, and an initial register binding $\rho_0: E_D \rightarrow R$, find a new register binding $\rho: E_D \rightarrow R$, such that the number of registers is minimized under a critical path delay constraint, where V_O is a set of vertices representing operations, E_O is a set of edges representing control dependencies (E_C) and data dependencies (E_D) between

operations; E_O is the union of E_C and E_D. Each element in E_D implies a data transfer from a source operation to a destination operation through a register (or a direct interconnect between chained operations), and data transfer between not-chained operations is to be bound to a register. F is a set of FUs, and R is a set of registers. As the *critical path delay constraint*, we use the initial critical path delay derived from ρ_0, which maps each data transfer to a dedicated register as shown in Figure 5.1(a) (we assume that the delay due to intra-partition interconnects and the capacitive loading by output registers is much smaller than the delay due to FUs and MUXs. Thus, we assume that the initial critical path delay is very close to the minimum achievable delay).

The register binding problem can be transformed to the minimum clique partitioning problem with constrained weight (MCPCW). [39] shows that this problem is NP-hard when the weight of a clique is represented by the sum of weights of vertices in the clique. Since the evaluation of the weight of a clique in our problem (i.e., evaluation of the critical path delay) is harder than that in MCPCW, the register binding problem is also NP-hard.

5.3 Proposed Algorithm

To solve the register binding problem, we devise a heuristic clique partitioning algorithm for a compatibility graph of data transfers, $G(V_V, E_V)$, where V_V is a set of vertices representing data transfers in a given CDFG, and E_V is a set of edges

connecting compatible vertices. Two vertices are compatible if the corresponding data transfers can share the same register, i.e. there is no overlap of live ranges between the data transfers, or they are from the same operation. The proposed algorithm iteratively constructs a maximal clique under critical path delay constraint. To construct the clique, it selects a vertex with minimum weight of merging. The weight $W(v,C)$ for merging a vertex $v \in V_V$ to clique C currently under construction is defined as follows:

$$W(v,C) = W_P(v,C) + W_S(v,C) + W_{DIST}(v,C) \tag{2}$$

$$W_P(v,C) = \begin{cases} 0, & F_s(v) \cap (\bigcup_{v_k \in C} F_s(v_k)) \neq \emptyset \\ 1, & otherwise \end{cases} \tag{3}$$

$$W_S(v,C) = \begin{cases} 0, & F_d(v) \cap (\bigcup_{v_k \in C} F_d(v_k)) \neq \emptyset \\ 1, & otherwise \end{cases} \tag{4}$$

$$W_{DIST}(v,C) = \begin{cases} \left|\bigcup_{v_k \in C^+} P_d(v_k)\right|, & \left|\bigcup_{v_k \in C^+} P_s(v_k)\right| = 1 \\ \infty, & otherwise \end{cases} \tag{5}$$

where $C^+ = \{v\} \cup C$, $F_s(v_k)$ is the set of the source FUs of v_k, $F_d(v_k)$ is the set of the destination FUs of v_k, $P_s(v_k)$ is the set of partitions that contain a source FU of v_k, and $P_d(v_k)$ is the set of partitions that contain a destination FU of v_k.

If a source FU of data transfer v is also a source of another data transfer already included in C, v can share the same input of the MUX in front of the shared register, so this merging does not increase the MUX delay. If a destination FU of data transfer v is also a destination of another data transfer already in C, then by

merging *v* with *C* (sharing the same register), *v* can also share the same input of the MUX in front of that FU, thus allowing this merging to also reduce the MUX delay. These binding aspects are considered in (3) and (4).

As shown in Figure 3.1, our architecture assumes that outputs of each partition are registered to avoid long combinatorial paths. This assumption also decreases the design space for register optimization since we are excluding the case of placing registers on the input side of FUs within a partition. Then data transfers from different partitions can never be grouped to share a register. Equation (5) prevents merging data transfer *v* to *C* when the source FU of *v* and the source FU(s) of *C* are in different partitions. As the number of partitions to which a register should provide data increases, the loading to the output of the register becomes larger; it should be discouraged. Equation (5) also reflects this as the cost of binding.

Although we use (2) for selecting data transfers to be merged, we do not use it for modeling the effect of register binding on the critical path delay since it does not accurately reflect the delay:

1) Even if the MUX size increases ($W_P(v,C)$ =1 or $W_S(v,C)$ = 1), the MUX delay may not increase in some cases. For example, when the number of inputs increases from three to four, the MUX delay may not increase since the height of the 2:1 MUX tree remains the same.

2) Although the number of partitions to which a register provides data can affect the interconnect length, the length can be short and thus ignored when those partitions are closely placed.
3) Merging of data transfers may increase the delay of some paths, but it does not always increase the critical path delay.

We devise an expression for T_{cp}, a better estimation of the critical path delay, as follows:

$$T_{cp} = max_{r_k \in R} T_R(r_k) \quad (6)$$

$$T_R(r_k) = max_{f \in pred(r_k)} T_F(f) + T_{MUX}(m(r_k)) \quad (7)$$

$$T_F(f) = max_{m \in M(f)} T_{MUX}(m) + T_{logic}(f) \quad (8)$$

where $T_R(r_k)$ represents the delay of the longest path from the controller to register r_k, $pred(r_k)$ is a set of FUs that provide data to register r_k, $T_F(f)$ represents the critical path delay of FU f (including the delay of its input MUXs), $m(r_k)$ is the MUX in front of register r_k, $T_{\mathrm{MUX}}(m)$ is the delay of MUX m, $M(f)$ is a set of input MUXs of f, and $T_{logic}(f)$ represent the logic delay of f itself. The MUX delay is estimated by the height of a 2:1 MUX tree; we do not differentiate 'delay from select input to data output of a 2:1 MUX' from 'delay from data input to data output' since they are almost the same according to our observation. T_{cp} obtained for the initial register binding is used as the critical path delay constraint (refer to Problem 2), and we

```
Register binding(V_V, E_V)
1   for each v_i in V_V
2       ρ_0(v_i) ← r_i
3   evaluate initial path delay T_cp(0) by (6)
4   n←0
5   while V_V ≠ ∅ {
6       calculate T_R(ρ_0(v)),  ∀ v ∈ V_V
7       C_n ← v with largest T_R(ρ_0(v))
8       U = {v ∈ V_V : v is adjacent to all vertices of C_n}
9       while U≠ ∅ {
10          update W(v_i,C_n),  ∀ v_i ∈ U
11          select v ∈ U with minimum cost W(v,C_n)
12          if T_R(C_n ∪ v) ≤ T_cp(0)  ∧  W(v,C_n) ≠∞ {
13              C_n←v,
14              U = U-{v_i ∈ U : v_i is not adjacent to v}
15          }
16          else {
17              U = U-{v}
18          }
19      }
20      V_V = V_V - C_n
21      n←n+1
22  }
```

Figure 5.2 Algorithm structure of register binding.

perform register binding such that all path delays do not exceed the constraint.

The proposed clique partitioning algorithm shown in Figure 5.2 iteratively constructs a maximal clique from a seed vertex while keeping the estimated delay (obtained by (7)) of the clique (register) under the critical path delay constraint.

More specifically, it first takes a seed vertex v_s that has the largest $T_R(\rho_0(v_s))$ as the initial clique. Then, it selects other vertices to be merged (bound) to the clique (register) based on the weights of the vertices in (2) (least-weighted vertex first). For each merge of a selected vertex, the algorithm evaluates the result by estimating the path delay using (7). If the path delay is equal to or shorter than the critical path delay constraint, it commits the merging. Otherwise, it restores the previous result. When adding more vertices to the clique is impossible, the clique is saved and the same process is repeated to construct another clique until all data transfers belong to their own clique. The proposed algorithm constructs a maximal clique by repeating the loop body starting at line 10 in Figure 5.2 at most $|V_V|$ times until U becomes null. Inside the loop body, since $T_R(r_k)$ can be calculted in $O(|V_V|)$ time and $W(v_i,C_n)$ can be updated in constant time, the complexity of constructing one maximal clique is $O(|V_V|^2)$. So, the complexity of the entire algorithm is $O(|V_V|^3)$.

This approach can incur register area overhead due to the policy of registering all outputs and sharing registers only on non-critical paths. However, it tends to reduce MUX area and delay. Additionally, the controller optimization flow, which will be explained next, reduces the number of buffer insertions during physical synthesis. So, the overall area overhead is tolerable when considering the improvement in clock period.

Chapter 6

Critical-Path-Aware Controller Optimization

6.1 Introduction

Datapath of target application is mostly implemented through the algorithms in previous chapters. Synthesis of controller is the other important work to make datapath to operate correctly and to optimize clock period since critical path mostly lies on the path from controller to datapath.

The controller, which is typically described by a finite state machine (FSM), has a significant impact on the performance of the synthesized hardware. Some researchers have contributed to controller optimization related to logic synthesis [40][41][42]. The state assignment and pipelining algorithm proposed in [40] optimizes the controller delay, which is measured from the latest FSM input to the

FSM output. [41] presents a method for partitioning and optimizing the controller in a hierarchical high-level description to reduce the implementation cost. Selection and hybrid approach between two possible FSM styles [42], Moore and synchronous Mealy machine, is proposed since they have the different area overhead and latency by the characteristic of application. However, these approaches focus on optimizing the control logic itself and do not consider the actual critical path from the controller to the datapath. [7] reports that a centralized controller worsens the critical path delay because of long wire length between the controller and datapath. To alleviate the problem, a distributed controller for RTL design is proposed. However, there is no consideration of HLS.

In this chapter, controller/MUX optimization method is proposed. Since logic delay is affected by output capacitive load, assigning capacitive load to its driver impacts on clock period. Proposed algorithm tries to assign high capacitive load to output registers of controller on the non-critical path. Since the organization of MUX tree which is driven by controller impacts on capacitive load to output registers, MUX optimization is also performed by encoding input selection signal of MUXs. Based on MUX encoding, control signals from the output register of controller are also encoded for correct operation of datapath.

6.2 Problem Formulation

As mentioned in Section 2, most critical paths are paths from the controller to data

registers. We can break a path delay down into various delay sources as follows.

$$T_{total} = T_{C2Q} + T_{logic} + T_{int} + T_{setup} \quad (9)$$

where T_{total} is the total path delay, T_{C2Q} is the clock-to-output delay of the register that starts the path, T_{logic} is the delay of the logical components including controller's output logic, MUXs, FUs, and buffers inserted to fix design rule violations, T_{int} is the interconnect delay, and T_{setup} is the setup time of the register at the end of the path.

Our overall objective is to minimize the maximum path delay, i.e., the minimum clock period. We consider that the setup time of a register is a fixed parameter, and MUXs and interconnects are optimized by the partitioning algorithm and the register binding algorithm as described in the previous sections. The minimum FU delay is also considered as a fixed parameter since the minimum critical path delay of an FU for a given technology library can be obtained before starting HLS steps. The controller optimization in this section focuses on optimizing the propagation delay of registers of the controller, inserted buffer delay, and the output logic delay of the controller. As explained in the previous section, a centralized controller should drive high load capacitance. This causes side effects during physical synthesis. Every technology library cell defines maximum capacitance and transition rules, which are easily violated in the controller. Physical synthesis tools typically fix the violations by inserting buffers when simple gate

sizing does not work, and thus, these tools inevitably add the delay of the inserted buffers to the paths. The propagation delay of a register (T_{C2Q}) is also modeled as a linear function of the load capacitance as follows when the capacitance is within a limited range [50].

$$T_{C2Q}(r) = \gamma \times l(r) + \delta \quad (10)$$

where $l(r)$ is the load capacitance driven by register r, and γ and δ are given in the library.

The objective of controller optimization in this work is to reduce the output logic delay and the load capacitance driven by the controller on the critical path. To identify the critical path, the path delays to registers are estimated by (9). Then, by using the algorithm that will be explained in the next subsection, we reduce the output logic delay and the load capacitance imposed on the controller that drives paths containing a critical FU and/or a long interconnect.

Figure 6.1 shows our motivational example for control optimization. Assume that the target application has been scheduled with four control steps. Also assume that the combinatorial path containing the multiplier is a critical path and combinatorial paths containing adders have slacks in time. Scheduling and binding results are shown in Figure 6.1(a), where "-" means don't care. Figure 6.1(b) shows a binary encoding of states, which is used to drive the MUXs with selection signals. As the control step advances, the value of the state register bit st_0 changes following

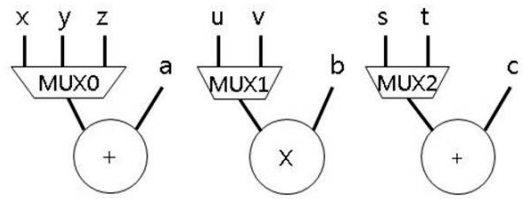
the pattern 0011, while st_1 changes following the pattern 0101. For the control of each MUX, the controller needs both st_0 and st_1 to generate a proper selection signal using some output logic, thus incurring unnecessary capacitive loading and logic delay.

We modify the controller such that the output registers (instead of state registers) can directly (without output logic) drive MUXs with proper control signals. Figure 6.1(c) shows one possible encoding of the output, where output register bits o_0 and o_1 can drive MUX0 directly, and o_2 can drive MUX1 and MUX2 directly with the required control signals. This approach can remove output logic delay of controllers and reduce the load capacitance imposed on the output registers.

Additionally, we can optimize the load capacitance by exploiting don't cares and by changing the order of MUX inputs as shown in Figure 6.1(d). Using these approaches, control signals to MUXs can be modified such that o_2 drives only MUX1. Because multipliers have long logic delay, this configuration of control signals can reduce the critical path delay.

Motivated by this example, we can define a controller optimization problem. Consider a set $P = \{p_1, p_2, \ldots, p_N\}$ of control patterns to control MUXs. If we have four control steps, for example, then 14 different patterns {0001, 0010, 0011, … 1110} are available. Note that patterns 0000 (always zero) and 1111 (always one) are excluded since they are useless. Therefore, the total number of possible patterns

	1+		2x		3+	
CS0	x	a	u	b	s	c
CS1	y	a	-	-	-	-
CS2	-	-	v	b	-	-
CS3	z	a	v	b	t	c



(a) Scheduling and binding results

	st_0	st_1	MUX0	MUX1	MUX2
CS0	0	0	x(00)	u(1)	s(0)
CS1	0	1	y(01)	-(0)	-(0)
CS2	1	0	-(10)	v(0)	-(0)
CS3	1	1	z(10)	v(0)	t(1)

	Load
st0	MUX0, MUX1, MUX2
st1	MUX0, MUX1, MUX2

(b) Binary encoding for state encoding

	o_0	o_1	o_2	MUX0	MUX1	MUX2
CS0	0	0	1	x(00)	u(1)	s(1)
CS1	0	1	0	y(01)	-(0)	-(0)
CS2	1	0	0	-(10)	v(0)	-(0)
CS3	1	0	0	z(10)	v(0)	t(0)

	Load
o_0	MUX0
o_1	MUX0
o_2	MUX1,MUX2

(c) Output encoding

	o_0	o_1	o_2	MUX0	MUX1	MUX2
CS0	0	1	0	x(01)	u(0)	s(1)
CS1	0	0	0	y(00)	-(0)	-(0)
CS2	0	0	1	-(00)	v(1)	-(0)
CS3	1	0	1	z(10)	v(1)	t(0)

	Load
o_0	MUX0
o_1	MUX0,MUX2
o_2	MUX1

(d) Exploiting don't cares and MUX inputs encoding

Figure 6.1 Examples of controller optimization.

(N_p) can be computed by:

$$N_p = 2^n - 2 \tag{11}$$

where n is the number of control steps. Then the problem of controller optimization

can be formulated as follows:

Problem 3: Given a set O of output register bits of a controller and a set P of possible patterns, find mappings $\sigma: M \rightarrow W(O)$ and $\tau: O \rightarrow P$ such that the critical path delay is minimized, where M is a set of MUXs and $W(O)$ is the power set of O. Thus, $\sigma(m)=w$, $w \in W(O)$, denotes that MUX $m \in M$ is controlled by the output register bits in w (note that for a k:1 MUX, $|w| = \lceil log_2 k \rceil$), and $\tau(o)=p$ denotes that the value of output register bit $o \in O$ changes according to pattern $p \in P$.

Controller/MUX optimization can be transformed to the column compaction problem where the weight of a column is the critical path delay. Since the column compaction problem is known to be NP-complete [43], the controller/MUX optimization problem is NP-hard.

6.3 Proposed Algorithm

To solve Problem 3, we first define a cost function that represents the critical path delay from output register bit o to data registers.

$$T_O(o) = T_{C2Q}(o) + max_{m,o \in \sigma(m)} T_M(m) + T_{setup} \quad (12)$$

$$T_M(m) = T_{MUX}(m) + T_{logic}(f(m)) + max_{r_k \in R(f(m))} T_{MUX}(m(r_k)) \quad (13)$$

where $f(m)$ denotes an FU connected to the output of MUX m. $R(f)$ denotes a set of registers that store the data from FU f. Note that (7) and (13) are different representations of the same path delay, except that (7) is the maximum path delay to

a given data register and (13) is the maximum path delay from a given input MUX.

The design space of the controller optimization problem is too large to explore for an exact solution. For each MUX *m*, we have a set *CP*(*m*) of candidate sets of control patterns for proper MUX selection inputs. *CP*(*m*) can be obtained by assigning proper control patterns to MUX selection inputs for care states and enumerating all different control patterns for don't-care states. Thus,

$$|CP(m)| = {}_{2^c}\mathrm{P}_k \times (2^c)^d \tag{14}$$

where *c* is the number of control bits (i.e., selection inputs) of *m*, *k* is the number of data inputs of *m* ($c = \lceil log_2 k \rceil$), and *d* is the number of don't-care states for MUX *m*. From these candidates, we should select a set of control patterns to minimize the critical path delay, which is a computationally very difficult problem. So, we adopt two heuristic algorithms: a greedy algorithm and a genetic algorithm.

Our greedy algorithm [38] to solve this problem is shown in Figure 6.2. In this algorithm, we focus on finding a mapping $\sigma: M \rightarrow W(O)$, while assuming that each output register bit of the controller is assigned with a unique control pattern to reduce the design complexity (i.e., $\tau: O \rightarrow P$ is a one-to-one mapping and pre-determined arbitrarily). This assumption can restrict opportunities to reduce the critical path delay further by duplicating output registers, which will be considered later in the genetic algorithm. The algorithm starts by finding a set *CP*(*m*) for each MUX *m* from *W*(*O*). Separately, we sort MUXs in descending order of $T_M(m)$, and

```
GreedySelect(St, I)
1  for each i in I
2      CSi ← subset S ⊂ St, when S can provide proper control to i
3  Sort I by T_cp(i)(descending) and by |CSi|(ascending) in case of tie
4  Initialize St
5  for each i in I {
6      mindelay ←∞
7      for each S in CSi {
8          if max_{st∈S} T_cp(st)< mindelay, when γ(i)=S {
9              γ(i)←S
10             mindelay←max_{st∈s} T_cp(st)
11         }
12         Update T_cp(st) and l(st), st∈ γ(i)
13     }
14 }
```

Figure 6.2 Algorithm structure of greedy controller optimization.

in the case of a tie, the MUXs are sorted in ascending order of the number of candidate sets of control patterns. The first ordering is to first consider the MUXs with long delay since they will significantly affect the final critical path delay. The second ordering is to first consider the MUXs with control inputs having fewer choices. Finally, in that order, the algorithm selects control patterns for MUXs in a greedy way. As shown in line 8 of Figure 6.2, we use the cost function defined by (12). The term T_{C2Q} is calculated by (10) and the term $max_{m,o\in\sigma(m)}\, T_M(m)$ is calculated by (13). T_{setup} is ignored since we assume that it is a constant parameter.

Although the proposed greedy algorithm can reduce the design space to be

explored, it is still unsuitable for examples with many control steps since the number of candidates $|CP(m)|$ increases exponentially. It also has the aforementioned limitation in optimizing the critical path delay due to prohibited duplication of an output register on the critical path. To overcome these limitations, we devise a genetic algorithm that optimizes the controller allowing duplication of output registers. Moreover, we use a more precise model of the MUX tree driven by the controller as shown in Figure 6.3(b). The model considers the variation of load capacitance of each selection input of the MUX. It also considers the variation of the internal delay from a selection input to the output of the MUX.

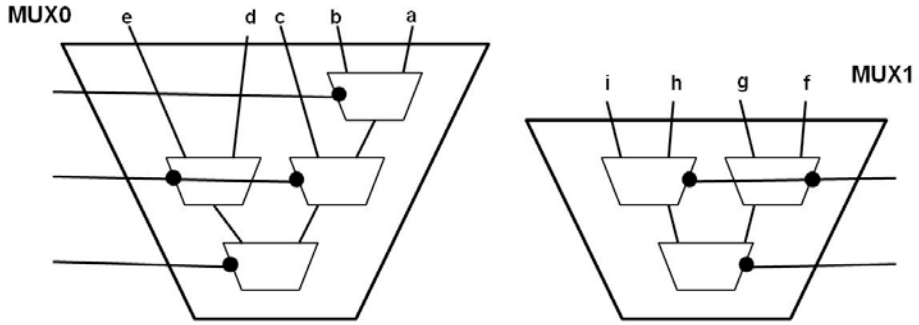
Figure 6.3 shows an encoding and evaluation example for a chromosome of the proposed genetic algorithm. Once scheduling and binding results are given, the chromosome of a MUX is created by encoding the string of control patterns, $\{p_0p_1p_2p_0p_2\}$, as shown in Figure 6.3(a). Design parameters are also given as shown on the right side of Figure 6.3(a); the row "*Register*" shows the parameters used in (10), "*T*" is the propagation delay of a 2:1 MUX, and "*cap.*" is the input capacitance. From the encoding, we can construct MUX trees for delay estimation as shown in Figure 6.3(b) using the given design parameters. In the fitness evaluation phase, we first estimate the minimum critical path delay as shown in Figure 6.3(c). For the estimation, each of the selection inputs of MUXs is assigned with its own output register. For a Pareto-optimal design, output registers are

	MUX0	MUX1
CS0	a(000)	f(00)
CS1	b(001)	g(01)
CS2	c(010)	-(00)
CS3	-(100)	h(10)
CS4	d(100)	-(10)
CS5	e(111)	i(11)
CS6	-(000)	-(00)

	MUX0			MUX1	
	p_0	p_1	p_2	p_0	p_2
CS0	0	0	0	0	0
CS1	0	0	1	0	1
CS2	0	1	0	0	0
CS3	1	0	0	1	0
CS4	1	0	0	1	0
CS5	1	1	1	1	1
CS6	0	0	0	0	0

Register	γ	25
	δ	0
2:1 MUX	T	20
	cap.	1
Remainder	T	200

(a)



(b)

Controller	
σ(MUX0) = {o_0, o_1, o_2}	τ(o_0) = p_0
	τ(o_1) = p_1
	τ(o_2) = p_2
σ(MUX1) = {o_3, o_4}	τ(o_3) = p_0
	τ(o_4) = p_2

	delay	critical path delay
$T_o(o_0)$	25*1+20*1+200 = 245	290
$T_o(o_1)$	25*2+20*2+200 = 290	
$T_o(o_2)$	25*1+20*3+200 = 285	
$T_o(o_3)$	25*1+20*1+200 = 245	
$T_o(o_4)$	25*2+20*2+200 = 290	

(c)

Controller	
σ(MUX0) = {o_0, o_1, o_2}	τ(o_0) = p_0
	τ(o_1) = p_1
σ(MUX1) = {o_0, o_4}	τ(o_2) = p_2
	τ(o_4) = p_2

	delay	critical path delay
$T_o(o_0)$	25*2+20*1+200 = 270	290
$T_o(o_1)$	25*2+20*2+200 = 290	
$T_o(o_2)$	25*1+20*3+200 = 285	
$T_o(o_4)$	25*2+20*2+200 = 290	

(d)

Figure 6.3 An example with genetic algorithm of controller optimization.

```
GeneticSelect
1   GenInitChromosomes (C, n)
2   while termination != true {
3   c0 ← Selection(C)
4   c1 ← Selection(C)
5   off ← Crossover(c0, c1)
6   off ← Mutation(off)
7   off ← Repair(off)
8   Evaluate(off)
9   Replace(C, off)
10      if (the best solution is not improved for 500 generation)
11          termination ← true
12 }
```

Figure 6.4 Algorithm structure of genetic controller optimization.

merged as shown in Figure 6.3(d). Two registers with the same control pattern are merged if merging them does not increase the critical path delay. For example, o_0 and o_3 are merged, but o_2 and o_4 are not merged.

We have implemented the genetic algorithm as shown in Figure 6.4. The "GenInitChromosomes" step generates initial chromosomes by randomly changing the encoding of MUX inputs as explained in Figure 6.3(a). Inside the loop, the algorithm selects two solutions (parents) using rank-based selection, makes one offspring with two-point crossover, and mutates the offspring with 2% of probability. Then it repairs the generated offspring to control MUX correctly. The cost of the offspring is evaluated by the cost of output registers under critical path

delay constraints as shown in Figure 6.3(c) and (d), and the offspring replaces one of the parents based on their costs. That makes a new generation. The process is repeated until the genetic algorithm is terminated when the solution no longer improves for 500 generations.

Chapter 7

Evaluation

7.1 Experimental Setup

We have implemented the proposed design flow in C++ language. The design flow starts with a behavioral description in the C language, which is first parsed and then optimized with the SUIF compiler [44]. From the SUIF intermediate form, a CDFG is generated using the CDFG library [45]. We have first performed scheduling and FU binding over the CDFG using an in-house tool. Then, for the centralized controller architecture, we have applied register binding using simulated annealing (SA) to minimize MUX area [9][46]. For a non-registered FSM controller, we have just performed state encoding to generate an RTL description to be used for logic synthesis, but for a registered FSM controller, we have also performed explicit encoding of MUX selection signals to connect register outputs directly to the

Table 7.1 Resource library (32bit)

Add.		**Sub.**		**MUX(2:1)**		
Delay(ps)	*Area(um^2)*	*Delay(ps)*	*Area(um^2)*	*Delay(ps)*	*Area(um^2)*	*Load(pF)*
140	370	130	420	20	90	0.032
SFT.		**Mul.**		**Reg**		
Delay(ps)	*Area(um^2)*	*Delay(ps)*	*Area(um^2)*	*T_{C2Q}-γ(ps/pF)*	*T_{C2Q}-δ(ps)*	*Area(um^2)*
130	290	460	2730	730	30	120

MUXs. For the proposed distributed controller architecture, we have applied the proposed algorithms. From the results, an RTL design with a centralized controller and the one with a distributed controller are generated. The RTL designs are fed to the Synopsys Design Compiler [48] to generate synthesized gate-level netlists, which are placed/routed using the Synopsys IC Compiler [49] with the TSMC 45nm nominal Vt technology library [50]. Table 7.1 shows parameters of the resource library that we have used for our design flow, where T_{C2Q}-γ and T_{C2Q}-δ are constants used in (10), and Load of MUX(2:1) is the total loading by the selection pin of a 32bit MUX.

Table 7.2 provides the details of benchmark examples used in this experiment and HLS results (scheduling and FU binding) of them. The examples include six realistic examples, including DCT from JM [51], FIR from DSP stone [52], FFT, product of matrix, 2D-convolution, and IDCT [53], and six synthetic examples; the synthetic examples are designed to present the effectiveness of our approach when design size increases. Since synthetic examples contain many operations and many

Table 7.2 Benchmarks details

Bench marks	**CDFG**		**Operations**				**Performance**		**Used Resources**			
	Node	*Edge*	*Mult.*	*Add*	*Sub.*	*Shift*	*II*	*CStep*	*Mult.*	*Add*	*Sub.*	*Shift*
SYN0	80	176	71	9	-	-	4	22	18	3	-	-
							-	12	7	1	-	-
SYN1	100	224	85	15	-	-	4	22	22	4	-	-
							-	11	8	3	-	-
SYN2	120	271	100	20	-	-	4	27	25	6	-	-
							-	12	9	3	-	-
SYN3	80	176	54	26	-	-	4	25	14	7	-	-
							-	12	5	3	-	-
SYN4	100	222	65	35	-	-	4	25	17	9	-	-
							-	11	7	4	-	-
SYN5	120	280	80	40	-	-	4	24	20	10	-	-
							-	12	7	5	-	-
DCT	60	129	16	13	13	18	4	24	4	4	4	6
							-	10	4	2	2	4
FIR32	64	129	32	31	-	1	4	17	8	8	-	1
							-	12	6	5	-	1
FFT	230	492	68	81	81	-	6	40	12	15	15	-
							-	16	6	8	8	-
PRODMAT	112	241	64	48	-	-	4	11	16	12	-	-
							-	12	7	11	-	-
CONV3X3	89	187	49	40	-	-	4	15	13	10	-	-
							-	13	6	7	-	-
IDCT	68	144	14	27	13	14	4	21	4	7	4	4
							-	10	3	4	4	4

of them are multiplication operations, they occupy a relatively large chip area and thus clearly show the effect of interconnect delay and load capacitance. Each benchmark has two rows in Table 7.2, where the upper one shows the result obtained by applying functional pipelining with an initiation interval given in the eighth column, and the lower one shows the result obtained without functional pipelining. Each row also shows the resource constraint given for the HLS. Based on the HLS results, we partition the datapath, bind registers, and optimize the controller.

The following abbreviations are used to represent the algorithms implemented for the synthesis steps:

1) Cent: centralized controller architecture with register binding using simulated annealing.
2) R-FSM: controller is implemented with registered FSM.
3) DC: datapath partitioning for distributed controller architecture.
4) CRB: critical-path-aware register binding.
5) Greedy: controller optimization based on the greedy algorithm.
6) Genetic: controller optimization based on the genetic algorithm.

7.2 Design Parameters and Computation Time

We have determined the number of partitions (k) by the policy presented in Section

Table 7.3 Design parameters and runtime

Bench marks	**Functional pipelining**	**Datapath Partitioning**		**Register Binding**		**Controller Optimization**			
		k	*Time (ms)*	*SA*	*CRB*	*Greedy*	*Genetic*		
				Time(ms)	*Time(ms)*	*Time(ms)*	*Pop.*	*Avg.Gen.*	*Time(ms)*
SYN0	w	10	0.21	95.5	9.0	0.24	100	1203.5	186.5
	w/o	4	0.044	47.4	6.3	114298	100	2830.75	273.5
SYN1	w	12	0.33	132.6	17.1	0.30	100	1330.8	251.7
	w/o	6	0.11	66.7	12.7	8153	100	4718	754.2
SYN2	w	16	0.52	169.9	28.1	0.34	100	1199.6	287.0
	w/o	8	0.17	73.7	20.1	174799	100	1973	411.2
SYN3	w	8	0.21	97.9	8.6	0.23	100	1698	315.3
	w/o	4	0.055	48.0	11.1	156673	100	4505	472.4
SYN4	w	12	0.42	138.1	17.0	0.30	100	1246.3	265.6
	w/o	6	0.15	66.5	11.9	31191	100	4925.2	843.5
SYN5	w	12	0.48	181.1	30.0	0.36	100	1645.8	501.1
	w/o	8	0.21	74.5	18.1	452291	100	2761.3	668.3
DCT	w	2	0.10	56.2	3.5	0.15	100	4613.5	387.3
	w/o	2	0.07	31.1	1.7	1812	100	5282.5	994.8
FIR32	w	4	0.46	56.9	1.6	0.17	100	3760.8	604.3
	w/o	4	0.43	43.9	1.6	79314	100	5238.3	830.5
FFT	w	16	0.13	456.2	153	14.5	100	5117.5	3574.4
	w/o	8	0.75	137.0	60.9	$2.9x10^7$	100	2649	3427.7
PRODMAT	w	12	0.66	128.0	6.4	0.49	100	1930.1	410.0
	w/o	6	0.50	78.7	4.4	82456	100	5448.2	1418.7
CONV3X3	w	8	0.32	101.6	4.5	0.43	100	2421.8	402.6
	w/o	4	0.17	66.5	2.9	668930	100	6173.3	1359.3
IDCT	w	4	0.21	78.3	3.3	0.32	100	3681.5	397.2
	w/o	2	0.19	39.4	2.0	355.7	100	6538.5	1093.4

4.4. Table 7.3 shows the value of design parameter k used for each benchmark and the computation time for each algorithm. Columns show respectively names of test examples, with or without functional pipelining, number of partitions, runtime of datapath partitioning, runtime of SA based register binding and CRB, runtime of Greedy controller optimization, and population size, average number of generations, and runtime for Genetic. For the average number of generations, we have averaged the number of generations over all partitions.

The runtime of CRB is much shorter than that of SA based register binding, while CRB outperforms SA based register binding in terms of critical path delay. The runtime of Greedy for designs using functional pipelining is very small, but that for designs not using functional pipelining increases exponentially. It is because $|CP(m)|$ increases exponentially as the number of control steps increases. Genetic provides acceptable computation time even for non-pipelined cases, which makes Greedy less competitive in terms of computation time.

7.3 Analysis Critical Path Delay on Distributed Controller Architecture

The distributed controller architecture obtained by the proposed datapath partitioning algorithm is more effective in reducing path delays from controllers to datapath than reducing path delays within datapath. Nevertheless, the delay cost function of the register binding and controller optimization focuses only on the

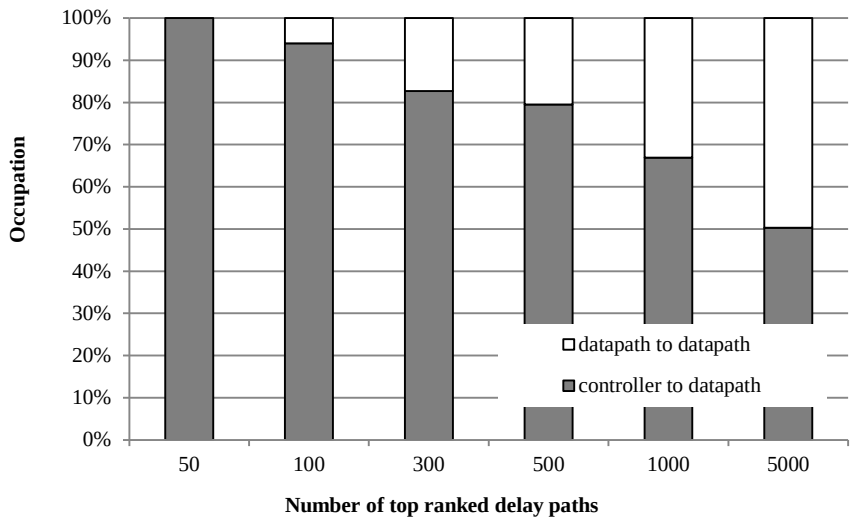


Figure 7.1 Analysis of critical path for distributed controller architecture.

paths from controllers to datapath, and thus one may have a concern that the paths within datapath become critical. However, the datapath partitioning algorithm actually penalizes partitioning that generates long interconnects, and register binding algorithm discourages the case where a data register provides data to many different partitions to suppress delay increases within datapath. Thus the path delays within datapath rarely dominate.

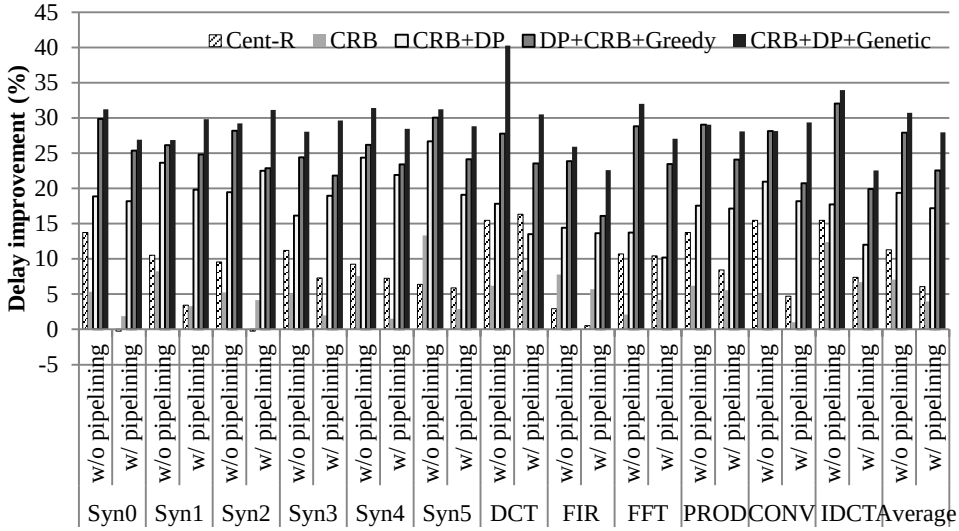
Figure 7.1 shows the results of timing analysis of the RTL circuit synthesized with a distributed controller for the same benchmark used for Figure 2.4. Compared to centralized controller architecture, more paths within datapath are included in the top-ranked delay paths, but delays of the paths from controllers to datapath still dominate. Communications between controllers may take longer. However, the path

delays between controllers are not critical since the involved logic delays are relatively small; those paths do not appear even among top 5000 longest paths for the benchmarks that we have used.

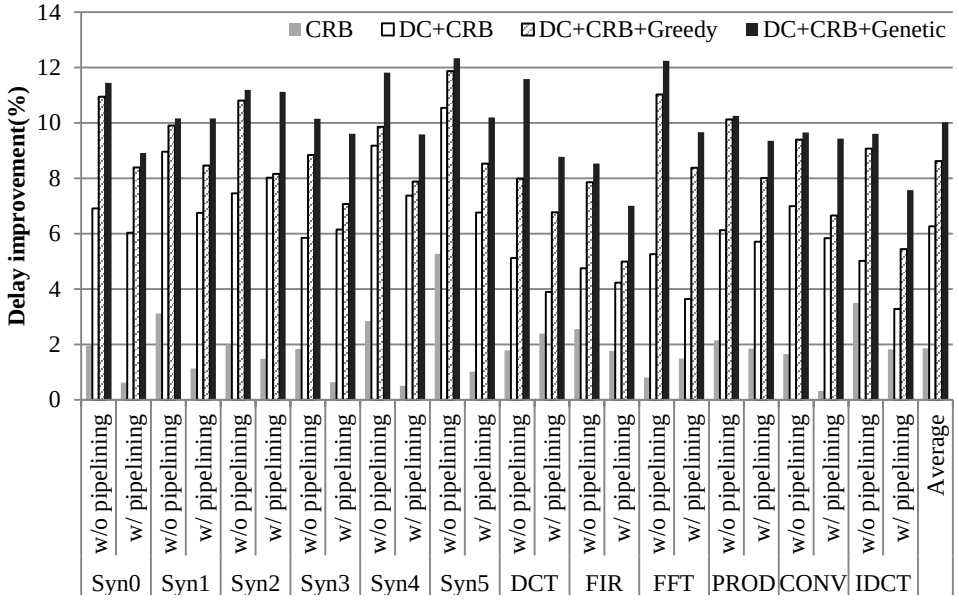
7.4 Analysis of Performance and Area

To show the effectiveness of our approach, reductions of critical path delay are depicted in Figure 7.2(a). Since FU delays and register setup time are given as fixed parameters, we exclude those delays. The delay values are normalized by the delays of the centralized controller architecture with non-registered FSM. Compared to it, our approach can reduce the sum of controller, MUX, and interconnect delays by 30.7% and 28.0% (in geometric mean) for non-pipelined and pipelined cases, respectively. If we use a registered FSM, we can reduce those delays even with the centralized controller architecture but only slightly (11.3% and 6.0% in geometric mean for non-pipelined and pipelined cases, respectively). In some examples (Syn0 and Syn2), especially for pipelined cases where the centralized controller should drive much more datapath components, the delay worsens compared to non-registered ones since the conventional registered FSM does not consider the capacitive loading to the output registers on critical paths.

CRB achieves delay reduction by reducing MUX delays on critical paths. It does not use MUXs to share registers on possible critical paths whereas conventional register binding algorithms tend to share registers even on the critical



(a) MUX, controller, and interconnect delay



(b) Critical path delay

Figure 7.2 Comparison results for performance.

path. Improvements on non-pipelined cases are more significant since MUXs in those cases tend to be larger than those in pipelined cases and they have much room for improvement in register binding. The datapath partitioning algorithm, which reduces interconnect delays and load capacitance driven by the controller, provides the most significant improvements among the three proposed steps. Both Greedy and Genetic reduce the critical path delay by assigning high load to controller output registers on non-critical paths and removing controller output logic circuits. Genetic reduces critical path delay more than Greedy. The source of improvement given by Genetic is exploring the design space for mapping controller output registers to control patterns, whereas Greedy assumes that each output register gives a unique control pattern. Thus Genetic can split a highly loaded output register to further improve the critical path delay.

Both datapath partitioning and controller/MUX optimization are redundant to optimize capacitive load. Figure 7.3 presents this aspect of two optimization flow. When datapath partitioning is applied first, improvement on datapath partitioning occupies 59% of total improvement. However, when controller/MUX optimization is applied first, improvement on datapath partitioning occupies only 25% of total improvement. It is because distributing capacitive load which can be acquired by datapath partitioning has already been acquired by controller/MUX optimization. Reducing interconnect delay from controller to datapath by datapath partitioning

occupies only 25 % of total improvement.

To analyze the source of improvement further, we break down the delay (minimum clock period) of each design by the component types, and presents buffer/inverter delay and register clock-to-output delay in Figure 7.4. Since our approach optimizes controller delay with distributed architecture and controller optimization algorithm, we compare cases: centralized architecture (Centralized) and proposed architecture with proposed algorithms (DC+CRB+Genetic). The main source of improvements of delay is the removal of buffers (including inverters) and reduction of the interconnect delay[8] on the critical path. Using distributed controller architecture and optimizing load capacitance driven by the controller allows removing buffers and reducing the delay of registers, and that of interconnects.

Considering that the controller, interconnect, and MUX delays in our benchmarks account for 35-40% of the total critical path delay when we use a centralized controller, the improvement in the total critical path delay obtained by our approach can be limited (note that our approach improves only controller, interconnect, and MUX delays). As shown in Figure 7.2(b), our approach reduces the total critical path delay by 10.0% on average, and such a reduction can alleviate

[8] In the timing report of the tool that we have used, the interconnect delay is included in the delay of the cell that drives the interconnect, and that is why we cannot see it in Figure 7.4.

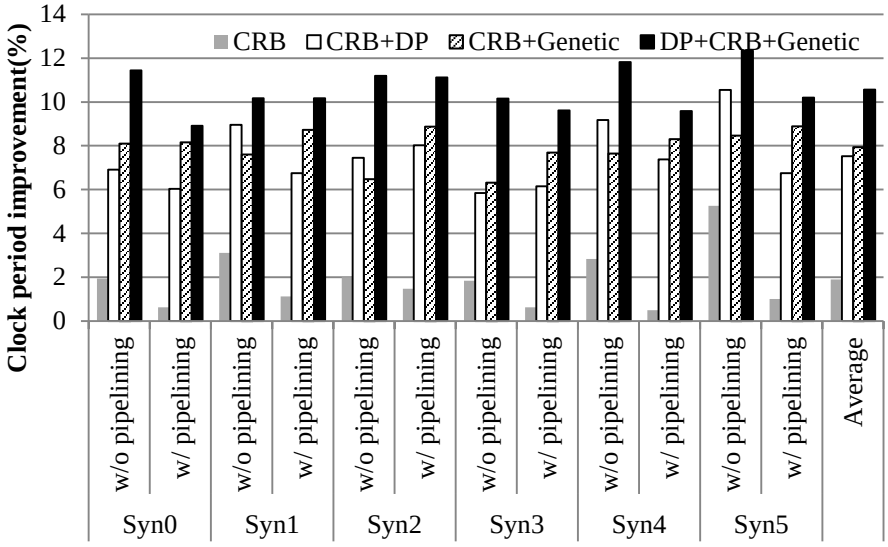


Figure 7.3 Optimization redundancy of datapath partitioning and controller/MUX optimization.

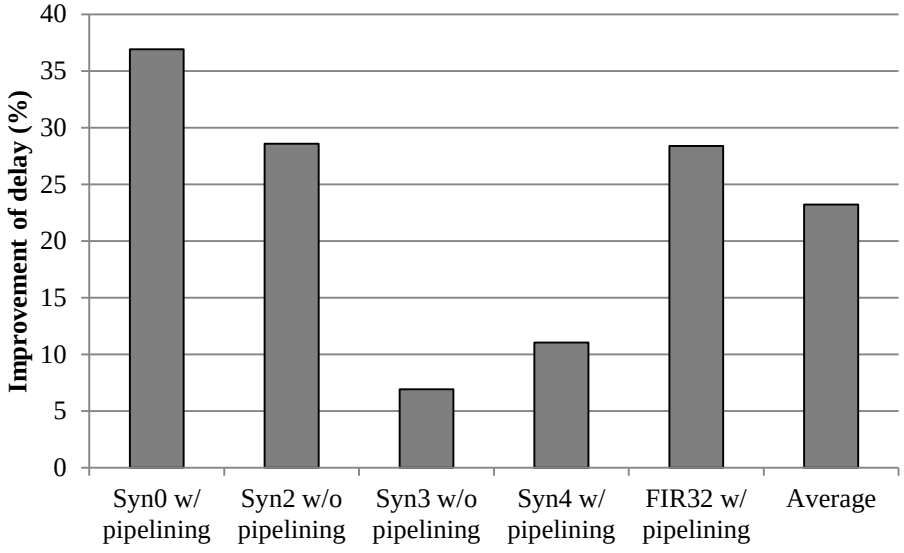


Figure 7.4 Improvement on buffer and register propagation delay.

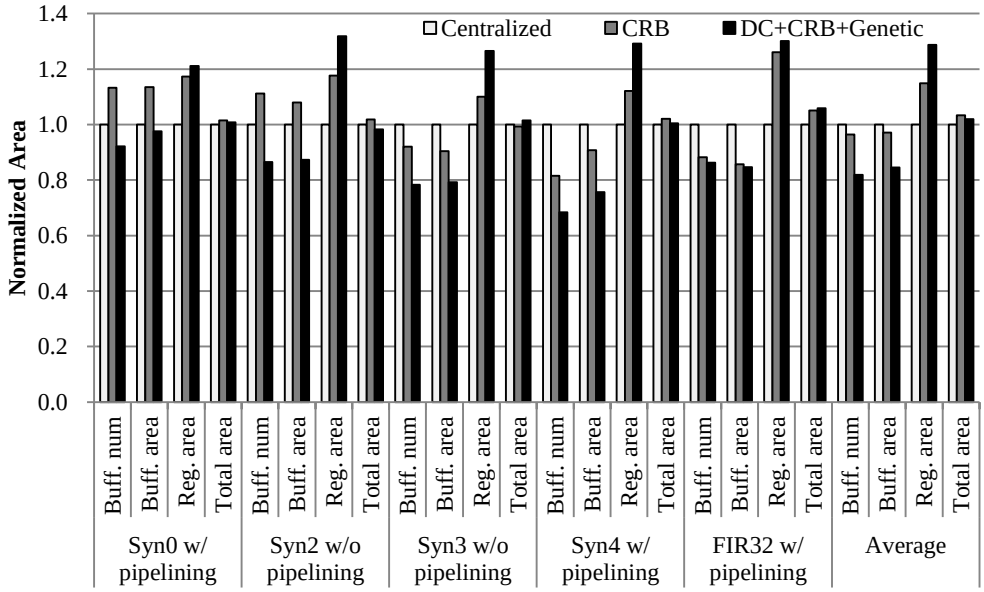


Figure 7.5 Comparison results for area.

timing closure problems effectively.

Our flow reduces the critical path delay at the cost of some area overhead, which is mainly caused by the increased number of register bits during register binding and controller optimization. Figure 7.5 shows the number and area of buffers/inverters, the area of registers, and total area normalized to centralized controller architecture. CRB increases the number of data registers since it does not share registers on critical paths. The datapath partitioning replicates controller, and the controller optimization replicates controller output registers, and thus they increase the number of registers. On the other hand, the number and area of buffers/inverters decrease when DC+Genetic is applied. Table 7.4 presents

controller information for the case of centralized controller (note that the controller is replicated for the distributed architecture). For example, the distributed controller architecture of FFT without functional pipelining has eight local controllers, each of which has 16 states. Column “Area” presents logic area of controller and the proportion of controller to total area. The fact that the controller is typically very small helps to reduce the overhead of controller replication of our approach. Therefore, the overall overhead is not serious as shown in Figure 7.5. Although our approach adds additional controller output registers and data registers to the circuit, combinational logic can decrease since additional data registers possibly remove register sharing MUXs, and distributed controller and controller optimization method can reduce buffer insertion and buffer sizing during physical synthesis. The overall area overhead of our approach is 2.2% on average. This overhead is significantly low compared to the performance improvement.

Physical synthesis tools typically allow improving performance at the cost of area. So, when the area of hardware generated by proposed approaches is restricted to the area of hardware generated by centralized controller architecture, performance improvement may be restricted. However, performance degradation by restricted area is limited only to 0.5% as shown in Figure 7.6.

Table 7.4 Information of controller

Bench marks	pipelining	# of states	Area (um^2 (%))	Bench marks	pipelining	# of states	Area (um^2 (%))
SYN0	w	4	284.4(0.3)	DCT	w	4	218.4(0.9)
	w/o	12	856.1(1.8)		w/o	10	520.7(1.4)
SYN1	w	4	362.5(0.3)	FIR32	w	4	171.1(0.4)
	w/o	11	944.1(1.7)		w/o	12	520.7(1.4)
SYN2	w	4	426.2(0.3)	FFT	w	6	1184.6(0.8)
	w/o	12	1171.6(1.7)		w/o	16	1841.3(2.6)
SYN3	w	4	286.7(0.3)	PRODMAT	w	4	328.5(0.3)
	w/o	12	848.0(2.1)		w/o	12	1014.3(1.9)
SYN4	w	4	330.9(0.3)	CONV3X3	w	4	282.1(0.3)
	w/o	11	996.7(1.8)		w/o	13	860.1(2.0)
SYN5	w	4	423.4(0.3)	IDCT	w	4	239.2(0.6)
	w/o	12	1171.6(1.9)		w/o	10	536.6(2.0)

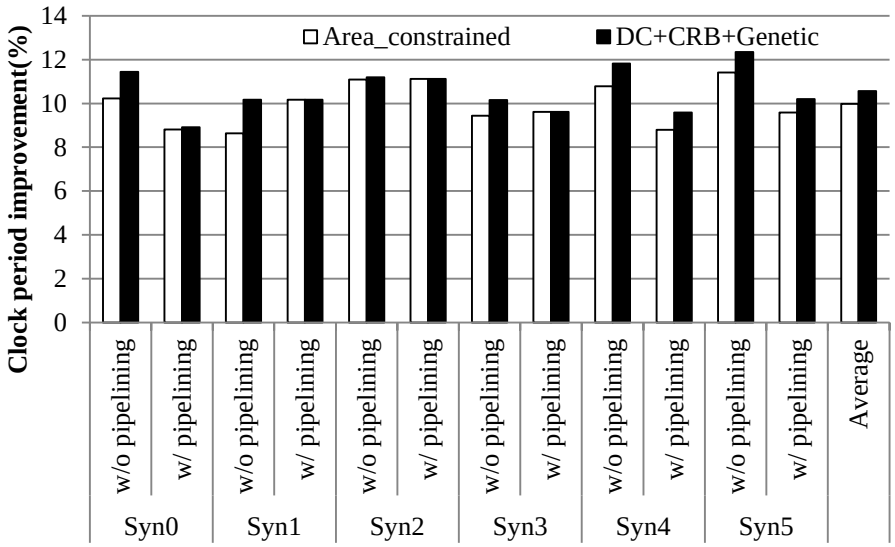


Figure 7.6 Performance improvement under area constraints.

7.5 Energy Consumption

Reducing energy consumption is very important issue on modern SoC design as well as HLS. Energy consumption is proportion to switching activity, effective capacitance, and supply voltage. Since the proposed method adds controllers and registers, the effective capacitance increases by the increase of area. Increase of register causes overhead of clock tree and internal power of register clock pin. On the other hand, the proposed method decreases total interconnect length, and energy consumption on interconnect decreases.

Figure 7.7 presents dynamic energy consumption which consists of cell internal energy and switching net energy. Cell internal energy, which is induced by short circuit current when switching cell, tends to increase since the number of registers increases and total area does. Switching net energy which is produced by driving current to drive output capacitance especially decreases for large example design since interconnect reduction is relatively significant for large one. Relation between total interconnect length and switching net energy is shown in Figure 7.8. Although decrease of total interconnect length and that of switching net energy do not match exactly, designs with significant decrease of interconnect length achieve lower switching net energy consumption.

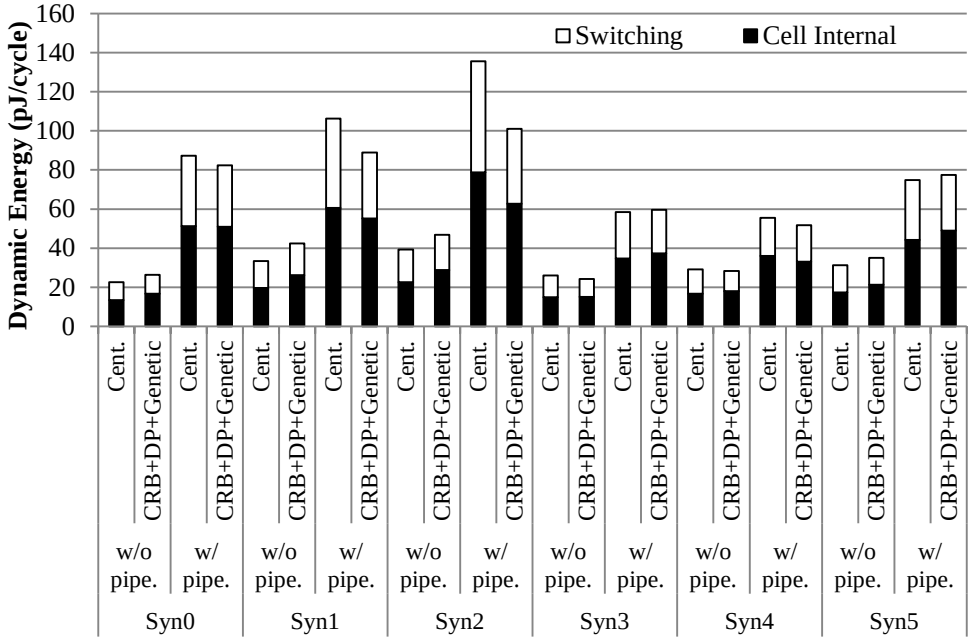


Figure 7.7 Dynamic energy consumption.

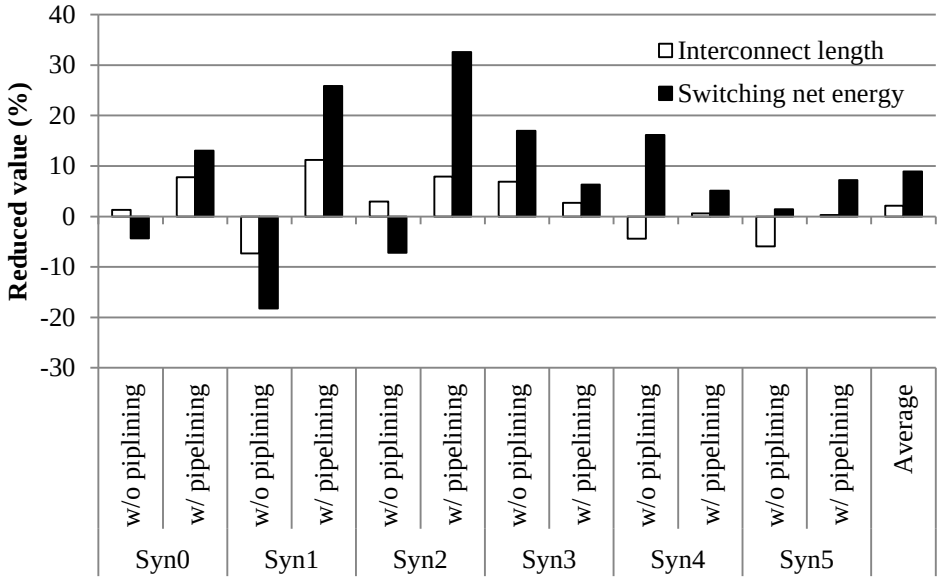


Figure 7.8 Interconnect length and switching net energy reduction compared to centralized controller architecture.

7.6 Analysis on Register Overhead

As shown in Section 7.4, the proposed method leads to register overhead. Register overhead makes clock tree larger, and it can cause poor routability of design, which makes clock period degradation and overhead in energy consumption, as well as area overhead. Section 7.4 and 7.5 present that those overheads from register increase can be compensated by improvements from proposed algorithms. However, the other problem, peak current overhead, may occur because of register overhead.

Registers can be the main source of current flow since register clock pins always switch at the same time during clock skew for each clock period while the other gates switch relatively intermittently. Register overhead causes larger peak current on the design. Since large peak current induces IR drop, it may affect the stability of system. Figure 7.9 presents increase of power consumption on clock network including register cell internal power on clock pins and peak power of designs acquired by Prime Time PX [54]. Although peak power is not exactly same as peak current, it is the best measure to reflect the variation of peak current in gate and layout level abstractions [55][56]. Proposed method increasing the number of registers consumes more not only clock network power but also peak power by about 25%. To alleviate peak current overhead, we can apply two approaches, clock gating approach and register constrained approach.

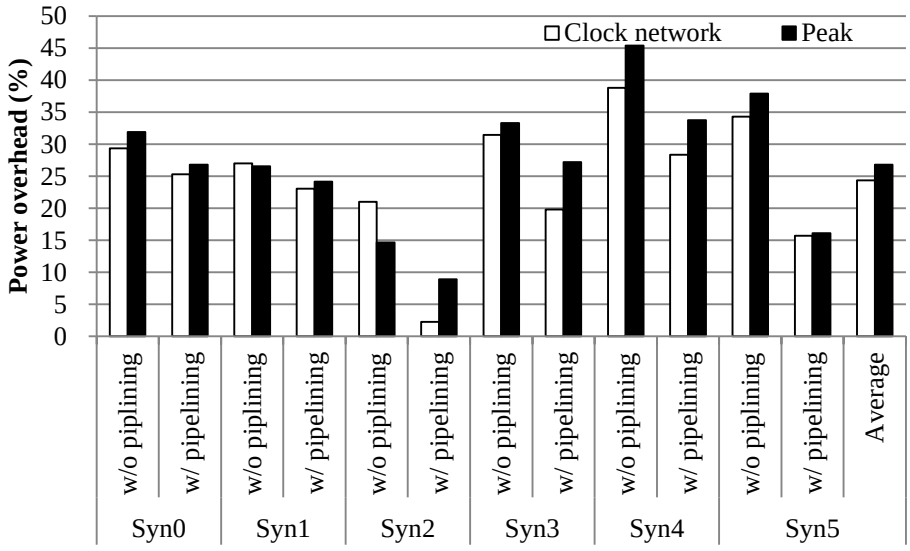


Figure 7.9 Clock network power and peak power consumption.

7.6.1 Clock Gating Approach

Figure 7.10 shows the reason why register overhead of proposed method increases peak current. Reg0 on the critical path splits to two registers Reg0 and Reg1 by proposed register binding algorithms to reduce critical path delay. Then, current flow in register clock pins become twice even when data to registers is not enabled. As shown in Figure 7.10(a), these registers are not concurrently enabled since they are separated from the same register. If we can block clock from the clock pin of register during the register is disabled, we do not suffer from peak current overhead from clock pin of register even though we use additional registers. Clock gating [57] is a popular technique among modern low power design methodology. It reduces cell internal power from register clock pin by gating clock with enable signal.

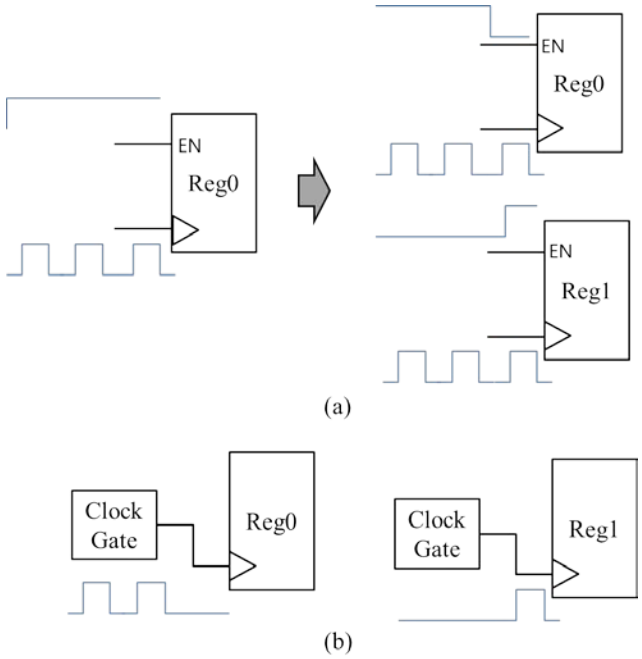


Figure 7.10 Clock gating: (a) peak current overhead from register overhead; (b) peak current reduction using clock gating.

Although it is not proposed to reduce peak current, we can utilize it to reduce peak current on the proposed method. Figure 7.10(b) presents an example of reducing peak current from clock. Since clock is gated by enable signal, only one register clock pin is switched for each clock cycle. So, we can reduce peak current from clock even though we use more registers.

To implement clock gating, we utilize automatic clock gating flow provided by Design Compiler in logic synthesis step. It replaces registers with enable signal to registers with clock gates. It also reduces clock gating overhead by sharing clock gates with the same enable signal. Result applying clock gating technique is

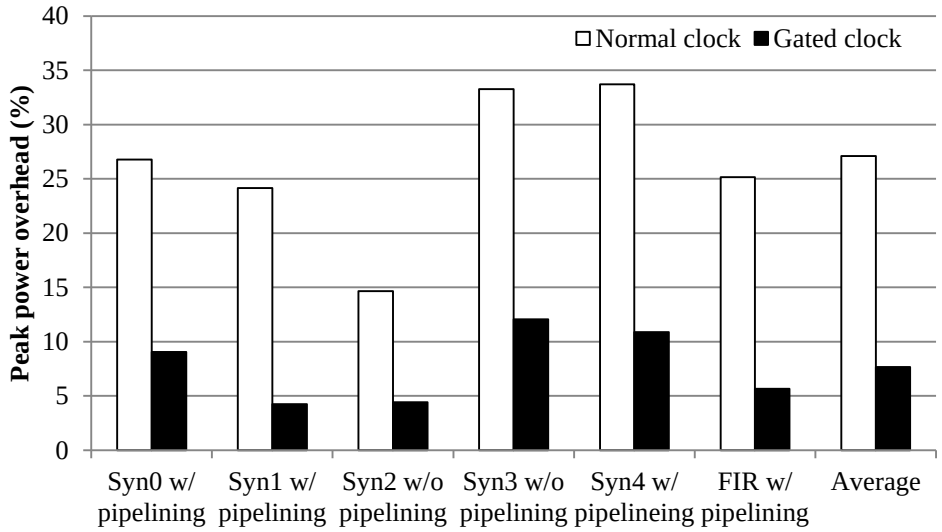


Figure 7.11 Reduction of peak power overhead using gated clock.

presented in Figure 7.11. It contains peak power overhead of proposed method compared to Cent. both without clock gating and with clock gating. Peak power increases only by 7.7% on average when clock gating is applied to proposed method while peak power increases by 27% on average proportion to increase of the number of registers when clock gating is not applied. 7.7% of peak power overhead is caused by large clock tree and many clock gates induced by register overhead. Binding data transfers from the same operation to different registers may also produce peak power overhead since these registers are clocked at the same time although they are split from the same register.

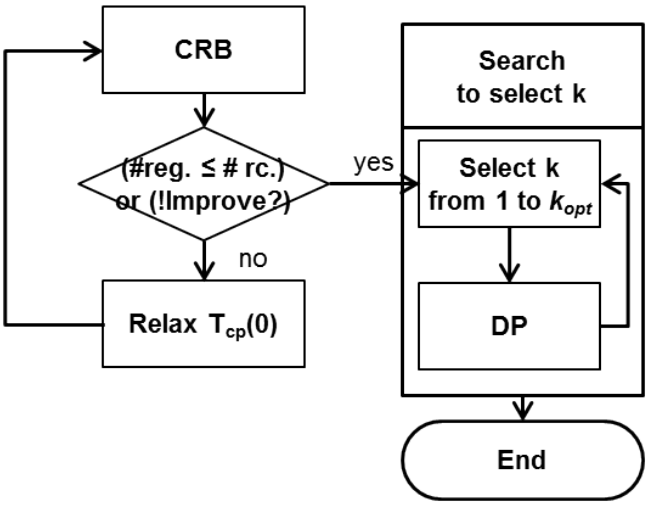


Figure 7.12 Modified flow with register constraint.

7.6.2 Register Constrained Approach

Although clock gating may be efficient to reduce peak power overhead induced by our approach without modifying the result from our approach, it has inherent overhead to insert clock gates to clock tree. In this section, we modify proposed datapath partitioning and register binding algorithms to reduce the register overhead.

The number of registers increases as the number of partitions increases since register sharing is restricted across different partitions. The proposed register binding algorithm also induces additional registers on the critical path since registers are added if path delay to register exceeds critical path delay constraint. Modified design flow presented in Figure 7.12 optimizes design under register

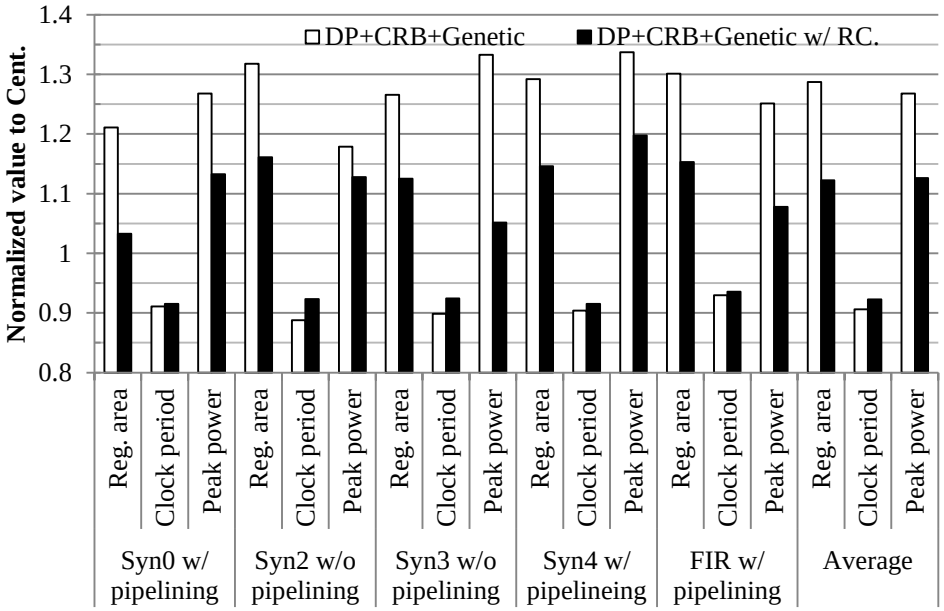


Figure 7.13 Performance, register area and peak power under data register overhead constraint by 15%.

constraint. At first, critical-path-aware register binding algorithm is performed to get the number of registers to be used. Critical path delay constraint is relaxed when the number of registers is more than register constraint, and these procedures are iterated until the number of registers is less than the register constraint. Then, the proposed flow finds the optimal number of partitions under register constraint in the range from 1 to k_{opt} which is acquired by the method in the Section 4.4.

Experimental results from the modified flow are presented in Figure 7.13. The total increase of registers is restricted in 16% when we give data register overhead constraint as 15% while proposed method without register constraint inflicts

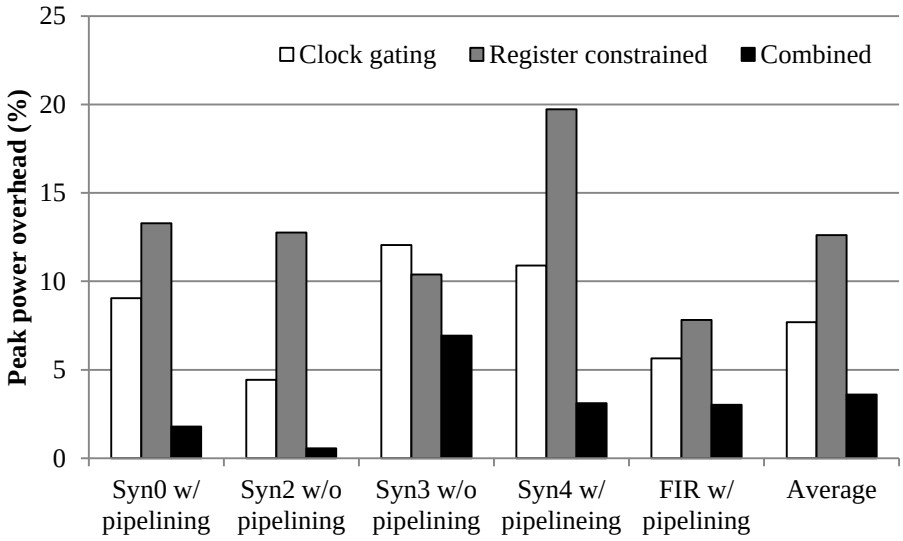


Figure 7.14 Combined approach to reduce peak power overhead.

register overhead up to 35%. Since adding registers and partitioning to improve clock period are restricted by register constraint, clock period improvement is degraded by about 1.6%. Peak power overhead is also reduced in proportion to the reduction of register overhead, and it is 12.7%.

7.6.3 Combined Approach

Reduction of register overhead by register constrained approach is limited to 16% since the first objective is minimizing path delay while conventional register binding minimizing the number of registers. However, clock gating presented in Section 7.6.1 can also be applied, and peak power can be improved further. Figure 7.14 presents peak power overhead of proposed approach which adopts both clock

gating and register constrained approach. Peak power overhead which reaches to 27% on average is suppress to 3.6% by using combined approach of clock gating and register constrained flow.

7.7 Join to Conventional Optimization Techniques on HLS

As explained in Section 3.2, our approach utilizes scheduling and binding results from conventional HLS flow. So, optimization techniques such as operation chaining including bit-level chaining and bit-width optimization can easily be applied for our approach. However, those techniques may affect the quality of results from our approaches.

For example, since chained operations have longer logic delay than not-chained operations, the portion of improvement, which proposed approach focuses on, is relatively reduced. However, capacitive load and interconnect may increase since chaining may restrict resource sharing and make design larger. Since area of design may smaller than design with uniform bit-width when bit-width optimization is applied, the effect of proposed method may be degraded. However, the portion of interconnect, MUX, and load capacitance may increases because of smaller FU delay, and improvement on critical path delay will increases.

7.8 Comparison with DRFM Binding Approach

The DRFM binding algorithm was developed in a recent research [32] for

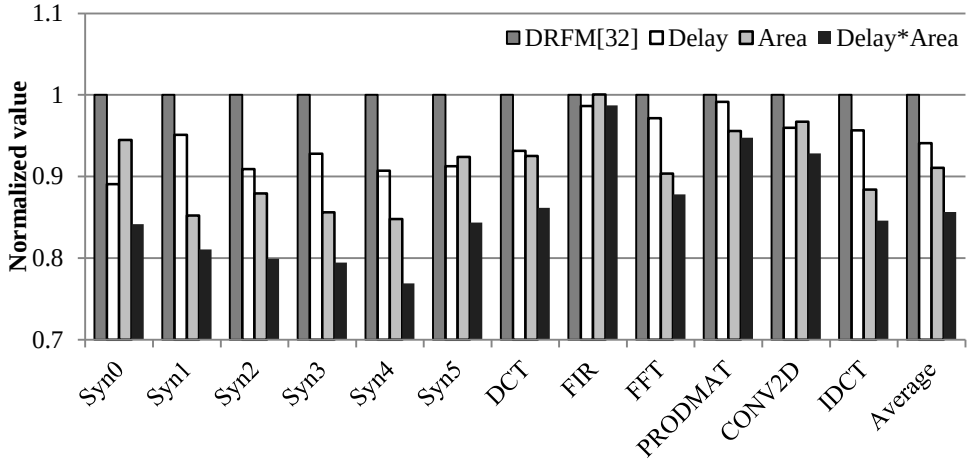


Figure 7.15 Comparison with DRFM.

distributed architecture. It optimizes the MUX delay and the number of global interconnects. Since it cannot handle the scheduling results with functional pipelining, we compare only for cases without functional pipelining as shown in Figure 7.15.

In terms of area-delay product, our approach outperforms DRFM architecture by 14.3% on average. This is because the DRFM binding algorithm focuses on reducing the average path delay by reducing the number of MUXs through the use of register files, and by reducing the number of global interconnects. On the other hand, our approach focuses on reducing candidate critical path delays with the critical-path-aware algorithm.

Chapter 8

Conclusion and Future Work

8.1 Summary

We analyzed the critical paths of typical designs with centralized controllers and observed that the critical paths arise on the path from the controller to data registers contrary to basic assumption of conventional HLS approaches. Based on this observation, we presented a hardware architecture with a distributed controller, and proposed a critical-path-aware HLS approach which integrated datapath and controller partitioning, register binding, and controller/MUX optimization. The datapath and controller partitioning tried to localize each of potentially critical interconnects within a partition or within a range of nearby partitions and to distribute capacitive load to controller. The register binding tried to reduce the MUX delay on potentially critical paths by sharing registers with MUXs only on

the non-critical path. The controller/MUX optimization tried to reduce the controller output logic and assign high load capacitance driven by the controller only on the non-critical path.

Experimental results showed that the proposed approach achieved 29.3% reduction on average in the controller, MUX, and interconnect delay with minimal area overhead. Also, the minimum clock period was reduced by 10.0% with 2.2% area overhead. Since proposed approach tried to reduce interconnect from controller to datapath, total interconnect may be reduced especially for large design. It provided reduction of dynamic energy consumption. Register overhead can cause peak current overhead, which may be the weakest point of proposed approaches. However, we proposed implementation level and algorithm level solutions to alleviate peak current overhead induced by register overhead. When compared to DRFM, a recently proposed distributed architecture, our approach outperformed by 14.3% in terms of delay and area product. We also propose two approaches, clock gating and register constrained flow, to alleviate high peak current problem which is caused by proposed approach. These approaches restrict peak current overhead fewer than 3.6%.

8.2 Future Work

There are several remaining issues as future work. As explained in Chapter 2, subtasks of HLS have interdependency with each other. Although proposed

algorithm gets results from scheduling and FU binding, it does not guarantee that given scheduling and binding results are also optimal after datapath partitioning, register binding, and controller/MUX optimization. An iterative approach, which makes up scheduling and binding results from the information provided by proposed algorithms such as long inter-partition interconnect, MUX delay, and controller delay, can help get more optimal solutions.

Interconnect delay becomes important for deep sub-micron technology. Many researches to estimate interconnect delay have done, but it remains that estimating individual interconnect delay exactly is very difficult compared to total interconnect estimation. As the cost function of proposed approach, individual interconnect delay estimation may be challenging and important future work for better quality of results.

Bibliography

[1] International Technology Roadmap for Semiconductors, http://www.itrs.net/

[2] IEEE Std.1666-2011, Standard for SystemC, IEEE Std. 1666, 2011.

[3] “Catapult,” http://www.calypto.com/

[4] “Cynthesizer,” http://www.forteds.com/

[5] “Synphony High-Level Synthesis,” http://www.synopsys.com/

[6] P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*, Springer Publishing Company, Inc., 2008.

[7] C. Papachristou and Y. Alzazefi, “A method of distributed controller design for RTL circuits,” in *Proceedings of Design, Automation, and Test in Europe*, pp. 774-775, Mar. 1999.

[8] A. Dasgupta and R. Karri, “Simultaneous scheduling and binding for power minimization during micro-architecture synthesis,” In

Proceedings of International Symposium on Low Power Electronics and Design, pp. 69-74, Apr. 1995.

[9] P. Kollig and B. M. Al-Hashimi, "Simultaneous scheduling, allocation and binding in high level synthesis," *Electronics Letters*, vol.33, no.18, pp. 1516-1518, 1997.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W,H.Freeman and Company, 1979.

[11] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, Inc., 1994.

[12] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASICs," *IEEE Trans. on Comput.-Aided Des. of Integr. Circuits and Syst.*, vol.8, no.6, pp. 661-679, 1989.

[13] N. Park and A. C. Parker, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications, " *IEEE Trans. on Comput.-Aided Des. of Integr. Circuits and Syst.*, vol. 7, No. 3, pp. 356-370, 1988.

[14] F. J. Kurdahi and A. C. Parker, "REAL: A Program for Register Allocation," In *Proceedings of Design Automation Conference*, pp. 210-215, June 1987.

[15] P. Brisk, F. Dabiri, R. Jafari, et al., “Optimal register sharing for high-level synthesis of SSA form programs,” *IEEE Trans. on Comput.-Aided Des. of Integr. Circuits and Syst.*, vol.25, no.5, pp. 772-779, 2006.

[16] C. Deming and J. Cong, “Register binding and port assignment for multiplexer optimization,” In *Proceedings of Asia South Pacific Design Automation Conference*, pp. 68-73, Jan. 2004.

[17] J. Cong and X. Junjuan, “Simultaneous FU and Register Binding Based on Network Flow Method,” In *Proceedings of Design, Automation and Test in Europe*, pp. 1057-1062, Mar. 2008.

[18] B. R. Rau and C. D. Glaeser, “Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing,” in *Proc. of Fourteenth Annual Workshop on Microprogramming*, pp. 183-198, Oct. 1981.

[19] B. R. Rau, “Iterative modulo scheduling: an algorithm for software pipelining loops,” in *Proceedings of the 27th annual international symposium on Microarchitecture*, pp. 63-74, Nov. 1994.

[20] R. Patasman, J. Lis, A. Nicolau, et al., "Percolation Based Synthesis," in *Proceedings of Design Automation Conference*, pp. 444-449, June, 1990.

[21] L.-F. Chao, A. S. LaPaugh, and Edwin Hsing-Mean Sha, "Rotation scheduling: A loop pipelining algorithm," *IEEE Trans. on Comput.-Aided Des. of Integr. Circuits and Syst.*, vol. 16, no. 3, pp. 229-239, 1997.

[22] S. Narayan and D. D. Gajski, “System Clock Estimation based on Clock Slack Minimization,” In *Proceedings of Design Automation Conference*, pp. 66-71, Sep. 1992.

[23] S. Park and K. Choi, “Latency minimization by system clock optimization,” *IEE Electronics Letters*, vol. 34, pp. 862-864, 1998.

[24] S. Bhattacharya, S. Dey, and F. Brglez, “Clock Period Optimization during Resource Sharing and Assignment,” In *Proceedings of Design Automation Conference*, pp.195-200, June 1994.

[25] J. Jeon, D. Kim, D. Shin, et al., “High-Level Synthesis under Multi-Cycle Interconnect Delay,” In *Proceedings of Asia South Pacific Design Automation Conference*, pp. 662-667, Feb. 2001.

[26] S. Park, K. Kim, H. Chang, et al., “Backward-annotation of post-layout delay information into high-level synthesis process for performance optimization,” In *Proceedings of 6th International Conference on VLSI and CAD*, pp. 25-28, Oct. 1999.

[27] Z. Gu, J. Wang, R. P. Dick, et al., “Unified incremental physical-level and high-level synthesis,” *IEEE Trans.Comput.-Aided Des. Integr. Circuits Syst.*, vol. 26, no. 9, pp. 1576-1588, 2007.

[28] V. Krishnan and S. Katkoori, “Clock period minimization with iterative binding based on stochastic wirelength estimation during high-level synthesis,” In *Proceedings of VLSI Design*, pp. 641-646, Jan. 2008.

[29] T. Kim and X. Liu, “A global interconnect reduction technique during high level synthesis,” In *Proceedings of the Asia South Pacific Design Automation Conference*, pp. 695-700, Jan. 2010.

[30] C. Huang, S. Ravi, A. Raghunathan, et al., “Generation of distributed logic-memory architectures through high-level synthesis,” *IEEE Trans.Comput.-Aided Des. Integr. Circuits Syst.*, vol. 24, no. 11, pp. 1694-1711, 2005.

[31] D. Kim, J. Jung, S. Lee, et al., “Behavior-to-placed RTL synthesis with performance-driven placement,” In *Proceedings of International Conference on Computer Aided Design*, pp. 320-325, Nov. 2001.

[32] J. Cong, Y. Fan, and J. Xu, “Simultaneous resource binding and interconnection optimization based on a distributed register-file microarchitecture,” *ACM Trans. Des. Automat. of Electron. Syst.*, vol. 14, no. 3, pp. 35-65, 2009.

[33] J. Cong, Y. Fan, G. Han, et al., “Architectural synthesis Integrated with global placement for multi-cycle communication,” In *Proceedings of International Conference on Computer Aided Design*, pp. 536- 543, Nov. 2003.

[34] A. Ohchi, N. Togawa, M. Yanagisawa, et al., “Performance-driven high-level synthesis with floorplan for GDR architectures and its evaluation,” In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 921-924, May 2010.

[35] S.-Y. Abe, M. Yanagisawa, and N. Togawa, “An Energy-efficient high-level synthesis algorithm for huddle-based distributed-register architecture,” In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 576-579, May 2012.

[36] C. Fiduccia and R. Mattheyses, “A linear time heuristic for improving network partitions,” In *Proceedings of Design Automation Conference*, pp. 175-181, June 1982.

[37] A. E. Dunlop and B. W. Kernighan, “A procedure for placement of standard-cell VLSI circuits,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 4, no. 1, pp. 92-98, 1985.

[38] S. Lee and K. Choi, “High-Level synthesis with distributed controller for fast timing closure,” In *Proceedings of International Conference on Computer Aided Design*, pp. 193-199, Nov. 2011.

[39] J. Li, M. Chen, J. Li, et al., “Minimum clique partition problem with constrained weight for interval graphs,” In *Proceedings of the 12th annual international conference on Computing and Combinatorics*, pp. 459-468, Aug. 2006.

[40] S. C-Y. Huang and W. H. Wolf, “Performance-driven synthesis in controller-datapath systems,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 2, no. 1, pp. 68-80, 1994.

[41] A. Seawright and W. Meyer, “Partitioning and optimizing controllers synthesized from hierarchical high-level descriptions,” In *Proceedings of Design Automation Conference*, pp. 770-775, June 1998.

[42] S. Park and K. Choi, "Sequential circuit optimization by FSM transformation," In *Proceedings of Asia Pacific Conference on Hardware Description Languages*, pp. 53-58, Jul. 1998.

[43] S. Mitra, L. J. Avra, and E. J. McCluskey, “An output encoding problem and a solution technique,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 18, no. 6, pp. 761-768, 1999.

[44] “The SUIF 1.x Compiler System,” http://suif.stanford.edu/suif/suif1/index.html

[45] J. Jeon, Y. Ahn, and K. Choi, *CDFG toolkit user's guide*, Tech. Rep. SNU-EE-TR-2002-8, Dept. Elect. Eng., Seoul National University, 2002. http://dal.snu.ac.kr/index.php/Software/CDFG

[46] A. Avakian and I. Ouaiss, “Optimizing register binding in FPGAs using simulated annealing,” In *Proceeding of international Conference on Reconfigurable Computing and FPGAs*, pp. 8-16, Sep. 2005.

[47] M. Celik, L. Pileggi, and A. Odabasioglu, *IC interconnect analysis*, Kluwer Academic Publishers, pp. 25-38, 2002.

[48] “Synopsys Design Compiler,” http://www.synopsys.com/

[49] “Synopsys IC Compiler,” http://www.synopsys.com/

[50] “TSMC Standard Cell Libraries,” http://www.synopsys.com/dw/tsmc.php

[51] G. K. Wallace, “The JPEG still picture compression standard,” *IEEE Trans. Consum. Electron.*, vol. 38, no. 1, pp. 18-34, 1992.

[52] V. Zivojnovic, J. Martinez, C. Schlger, et al., “DSPstone: A DSP-oriented benchmarking methodology,” In *Proceedings of International Conference on Signal Processing Applications and Technology*, pp. 715-720, Oct. 1994.

[53] “GAUT – High-level synthesis tool,” http://hls-labsticc.univ-ubs.fr/

[54] “Prime Time PX,” http://www.synopsys.com/

[55] S.-H. Huang, C.-M. Chang, and Yow-Tyng Nieh, “State Re-Encoding for Peak Current Minimization,” In *Proceedings of International Conference on Computer-Aided Design*, pp. 33-38, Nov. 2006.

[56] J. Gu, G. Qu, L. Yuan, et al., “Peak current reduction by simultaneous state replication and re-encoding,” In *Proceedings of International Conference on Computer-Aided Design*, pp. 592-595, Nov. 2010.

[57] G. K. Yeap, *Practical Low-Power Digital VLSI Design*, Kluwer Publishing, 1998.

한글 초록

공정기술의 급속한 발전으로 인해, 소비자의 다양한 욕구를 반영하기 위한 기능들이 하나의 칩에 집적되는데 반해 시스템 설계자의 생산성은 매우 더디게 발전하고 있다. 따라서 설계 과정에서 더 높은 수준의 추상화를 사용하는 것이 설계 시간 및 비용을 감소시키고 최적의 설계를 찾아 내기 위해 중요한 방법이 되고 있다. 행위 기술 모델로부터 레지스터 전송 모델을 설계해주는 상위수준 합성은 설계 생산성을 향상시키기 위한 연구 분야에서 중요한 주제가 되어 왔다. 상위 수준 합성에서 주로 사용하는 중앙 집중형 제어기의 경우 긴 연결선과 큰 정전용량을 야기해서 임계경로가 제어기에서 데이터패스 사이에서 주로 나타난다. 그러나 일반적인 상위 수준 합성에서는 데이터패스 내부의 지연시간만을 고려하기 때문에 실제 칩으로의 구현과정에서 성능제약 조건을 만족시키기 어렵게 한다. 따라서 본 논문에서는 이러한 문제를 해결하기 위해서 분산형 제어기를 사용하는 하드웨어 구조와 임계경로를 고려하는 상위 수준 합성 방법을 제안한다. 제안하는 방법은 데이터패스 분할, 레지스터 할당, 제어기 최적화 방법을 포함하며, 사용하는 하드웨어 구조를 최적화하기 위한 주요 변수인 분할 개수에 대한 설계 공간 탐색을 수행한다. 이를 통해서 제안한 방법은 기존의 중앙 집중형 하드웨어 구조에서의 상위 수준 방법에 비해, 2.2% 정도의 면적

비용으로 10%의 성능 개선을 얻을 수 있었다. 또한 제안한 방법에 야기할 수 있는 가장 큰 문제인 최대 전류 증가를 해결하기 위한 방법을 제안하여, 최대 전류의 증가량이 3.6%가 넘지 않도록 제한할 수 있었다.

주요어 : 상위 수준 합성, 분산형 제어기 구조, 레지스터 할당, 제어기 최적화

학번 : 2008-30236

감사의 글

지난 6년간의 학업을 박사논문으로 정리하면서 내용의 미흡함에 하루에도 여러 번 아쉬움이 남습니다. 이렇게 미흡한 논문이지만 온전히 제 힘만으로는 이룰 수 없었기에 그 동안 도움을 주신 많은 분들께 이 글을 통해 감사의 마음을 전하려고 합니다.

먼저, 지난 8년동안 저를 이끌어주신 최기영 교수님께 깊은 감사를 드립니다. 저의 연구가 방향을 잃지 않도록 조언과 지도를 해 주신 덕분에 조금이나마 학문적 성과를 이룰 수 있었습니다. 그리고 교수님께서 항상 학생들에게 보여주시는 학문에 대한 열정과 인간적인 배려는 제가 앞으로 살아가면서 큰 귀감이 될 것 같습니다. 그리고 바쁘신 와중에도 논문 심사에 참여해 주시고 좋은 학위 논문이 될 수 있게 많은 조언을 해주셨던 채수익 교수님, 김태환 교수님, 하순회 교수님, 이강희 박사님께 감사의 말씀을 전합니다.

석/박사 과정을 보내면서 많은 설계자동화 연구실 선후배 동료들과 좋은 일, 힘든 일을 함께 했습니다. 특히, 석/박사 과정에서 많은 연구를 함께 했던 이강희 박사님, 그리고 저의 박사 과정 동안 함께 상위수준 합성에 대한 연구를 하고 석사로 졸업한 동엽이와 재훈이형 덕분에 제 연구분야에 대해 조금 더 깊이 있는 이해를 할 수 있게 되었습니다. 그리고 저의 대학원 생활 모두를 함께 보낸 기성이형, 현직이형, 임용이, 이제 함께 사회로 나갈 만휘, 규승이, 학림이, 앞으로 고생할, 그리고 일 복

터진 후배 박사과정들 한민이, 진호, 경훈이형, 준환이, 동우, 재민이, 이제는 중요한 선택의 기로에 서있을 석사과정 선욱이, 성주, 남형이, 그리고 먼 한국까지 와서 열심히 공부하고 있는 Pierre, 선후배 동료들 모두 감사합니다. 좋은 선후배 동료들이 있었기 때문에 힘든 대학원 과정을 즐겁게 보낼 수 있었습니다. 비록 지금은 제가 먼저 학교를 떠나지만 하시는 연구는 물론 다른 일들도 모두 잘 되셨으면 좋겠습니다. 그리고 각자의 영역에서 최선을 다하며 기쁜 일이나 슬픈 일 모두 함께 나누고 응원해주는 대학동기, 친구들에게 감사함을 전하고 싶습니다.

가족들의 도움과 응원 없이는 기나긴 학업을 잘 견뎌내기 힘들었을 것 같습니다. 아직 미래가 불투명한 박사과정인 저를 믿고 결혼을 허락해주시고 물심양면으로 지원을 아끼지 않으신 장인어른, 장모님께 깊은 감사를 드립니다. 저에게 결혼과 아이라는 축복을 안겨주고 힘든 시간을 함께 해 준, 언제나 저를 믿고 지지해 주는 아내 희경이에게 사랑한다는 말을 전합니다. 가끔은 새벽에 아빠를 힘들게 하지만 언제나 큰 웃음을 안겨주는 사랑하는 종하, 건강하게 자라주면 좋겠습니다. 이제 새로운 시작을 준비하고 있는 동생 석영이, 형이 언제나 응원하고 있는 것을 잊지 않았으면 좋겠습니다. 무엇보다도 제 결정을 항상 믿고 지원해주시는 부모님께서 안 계셨다면 오늘의 저는 있지 않았을 것입니다. 사랑하는 부모님, 항상 건강하시길 바랍니다. 그리고 감사합니다.